IN THE UNITED STATES
PATENT AND TRADEMARK OFFICE

In re patent of:

Thomas Sean Houlihane
Appl. No.: 10/743,473 (Pub. No.: US 2005/0004777)
Publication Date: Jan. 6, 2005 – **Applicant's amended claims dated: January 22, 2007**
For: GENERATION OF A TESTBENCH FOR A REPRESENTATION OF A DEVICE

Submission of Prior Art Under 37 CFR 1.501

To:

Mail Stop M Correspondence
Hon. Commissioner for Patents
P.O. Box 1450
Alexandria, VA 22313-1450

Sir or Madam:

The undersigned herewith submits in the above identified patent the following prior art
(including paper copies and electronic duplicates thereof) which is pertinent and applicable to the
patent and is believed to have a bearing on the patentability of at least claims 1, 3-25, 27-46:

1. A copy of presentation slides entitled: <u>System-on-chip (SOC) and its Effect on
   Microcomputer Bus Products,</u> as presented to the VMEbus Standards Organization
   (VSO) on July 21, 1999 in Vancouver, B.C. Canada.

   The document provided is original except for minor formatting changes required to print
   it out using a current version of Microsoft PowerPoint, and changes to the copyright
   notice allowing unlimited copying and distribution. 31 pages.

2. A copy of a technical reference manual entitled <u>VME64 to PCI Bridge System-on-Chip
   (SoC)</u>, first published by Silicore Corporation on December 7, 2002.

   The document provided had changes made to the copyright notice and company address
   in 2004 (see p. 4). 129 pages.

(Page 1 of 6)

THIS PAGE BLANK (USPTO)

3. A copy of the VHDL source code files representing the VMEcore entity and testbench, as produced by Silicore Corporation's parametric core generator known as The VMEbus Interface Writer™ on 14 Feb 2002.

    Permission is hereby granted by Silicore Corporation to disregard the copyright notice listed at the top of each file page and substitute it with: "Copyright © 2002 Silicore Corporation. This document is provided under the terms of the GNU Lesser GPL License 2.1 as published by the Free Software Association." 46 pages.

4. A copy of the VHDL source code files representing the VMEcore entity as distributed with the VME64 to PCI Bridge System-on-Chip (SoC), and described at § 4.7.3 *VHDL Synthesis and Test* (p. 81) in the technical reference manual. 31 pages.

5. A copy of WISHBONE System-on-Chip (SoC) Interconnection Architecture for Portable IP Cores, Revsion B.2, dated: October 10, 2001. 109 pages.

6. A copy of a press release from Altium Ltd. (Sydney, Austrailia) dated 28 Apr 2003 entitled: Altium Unveils new 'Board-on-Chip' Technology. 3 pages.

    This file was downloaded on 21 Mar 2007 from URL:
    http://www.altium.com/files/corp/media/pdfs/mr_corp_280403.pdf

7. A copy of a press release from Altium Ltd. (Sydney, Austrailia) dated 17 Nov 2003 entitled: Altium Introduces Systems Focus to FPGA Design with Nexar. 5 pages.

    This file was downloaded on 21 Mar 2007 from URL:
    http://www.altium.com/files/corp/media/pdfs/mr_171103.pdf

Each of these references discloses a method for the GENERATION OF A TESTBENCH FOR A REPRESENTATION OF A DEVICE. It is believed that each of these references fall within the scope of Houlihane (as defined by the specification), and therefore has a bearing on the newly proposed claims.

Item 1, System-on-chip (SOC) and its Effect on Microcomputer Bus Products, is a presentation used to publicly announce VMEcore™, an IP Core that operates as a bus interconnect block advantageously interfacing the VMEbus microcomputer bus to WISHBONE, a System-on-Chip residing inside of a semiconductor integrated circuit. With respect to this document:

a) The announcement starts on page 19 of the presentation. VMEcore is a representation of a device, written in the VHDL hardware description language, which is incorporated into a data processing apparatus.

b) VMEcore is created by a software processing tool called <u>The VMEbus Interface Writer</u>™, a parametric core generator that: (i) receives configuration data options in the form of a machine readable text file, (ii) autonomously generates a VHDL testbench that makes reference to the configuration data, and which includes a first set of templates, known as test vectors defining inputs and outputs in a test environment and (iii) autonomously generates a representation of the device with reference to the configuration data in the form of a second set of templates including in the form of one or more synthesizable VHDL files.

c) At p. 24 in the presentation, <u>The VMEbus Interface Writer</u> was demonstrated to the public audience at the July 21, 1999 VSO meeting, and included: (i) the examination of all input and output files and (ii) operation in 'batch mode' where the tool is operated in combination with other tools as defined in a script file. This demonstration represents the last possible critical date for *first public use* and *first offer for sale* of this technology.

d) The internal operation, methods for combining IP cores and test elements, and related documentation for <u>The VMEbus Interface Writer</u> is a trade secret of Silicore Corporation, and have never been publicly displayed.

Item 2, <u>VME64 to PCI Bridge System-on-Chip (SoC)</u>, is the technical reference manual for a VMEbus to PCI bus bridging System-on-Chip that uses a VMEcore™ interface as created by <u>The VMEbus Interface Writer</u>™ and which was successfully reduced to practice. With respect to this document:

a) The detailed operation of the interface is described in *§ 4.7 VMEcore™ Entity* (p. 81-101).

b) A block diagram showing the operation of VMEcore with respect to other system elements is shown on p. 5.

c) This SoC was created with U.S. Government funds, with original documents available to U.S. Government employees under the contract number shown on p. 2. It was originally developed and sold directly to the U.S. Army Crusader project, a 155 mm self-propelled Howitzer cancelled by the Bush Administration in 2002. The Crusader has similar features to Archer, a 155 mm self-propelled Howitzer made by BAE Systems / Bofors (Sweden).

Item 3 is a compilation of VHDL and text files autonomously submitted to, and obtained from, <u>The VMEbus Interface Writer</u> in order to create a custom copy of VMEcore used in the system described in Item 2. With respect to these files:

a) File <SlavTest.txt> contains configuration data supplied to <u>The VMEbus Interface Writer</u> which describes the precise parameters that it should use to create the customized VMEcore output files, including a representation of the device. For example, the line

specifying "SLAVE: A24:SGL" indicates that the tool should create a VMEbus interface that supports the A24 addressing mode. Similarly, lines beginning with a ';' character are commented out.

b) File <SlavTest.log> is an output file indicating that the batch operation worked successfully, and includes a time and date stamp.

c) File <VmeCoreT.vhd> is an output file which is a testbench that makes reference to configuration data near the comment field marked "Interface:" (e.g. "A24:SGL"). The body of the file contains a first set of VHDL templates which define the test environment, including the ability to read a test vector file located in file <VmeCoreV.txt>. This operation is performed using a variety of standard, IEEE compliant VHDL simulation tools available from a number of suppliers.

d) The remainder of the files have a '.vhd' file extension indicating that they are VHDL files, and which are advantageously arranged according to the hierarchical diagram shown in Figure 4-29 (p. 89) of the technical reference manual listed in Item 2. This second set of templates each make reference to the configuration data near the comment field marked "Interface:" (e.g. "A24:SGL").

Item 4 is a compilation of files included with the standard distribution of the VME64 to PCI Bridge System-on-Chip (SoC). With respect to these files:

a) This compilation of files was adapted from the standard VMEcore output described in Item 3 above to: (i) include a 'DDIR' signal which was requested by the customer on or about 1 Mar 2002; (ii) by changing the copyright notice on 17 Jan 2004; and (iii) by eliminating the testbench and test vectors files which were not needed for the high level, hierarchical bridge SoC design (a separate file, not shown here, is included for that purpose).

b) These changes were implemented by modifying the output files rather than The VMEbus Interface Writer tool itself. This prevented the use of U.S. Government funds when modifying the tool, thereby preventing it from being subjected to U.S. Government Purpose Rights under DFARs 252.227-7014 (JUN1995), including inspection requirements thereof.

Item 5, WISHBONE System-on-Chip (SoC) Interconnection Architecture for Portable IP Cores, Rev. B.2, defines the interconnection system advantageously used by the aforementioned methods. With respect to this document:

a) From the Introduction on p. 7: "This specification does not require the use of specific development tools or target hardware. Furthermore, it is fully compliant with virtually all logic synthesis tools. However, the examples presented in the specification do use the VHDL hardware description language. These are presented only as a convenience to the

reader, and should be readily understood by users of other hardware description languages (such as Verilog®). Schematic based tools can also be used."

b) A specific objective of the WISHBONE SoC indicated on p. 10 is: "A further objective of the specification is to be independent of the underlying hardware description. For example, soft cores may be written and synthesized in VHDL, Verilog® or some other hardware description language. Schematic entry may also be used."

c) A specific objective of the WISHBONE SoC indicated on p. 11 is: "A further objective is to create an architecture that fully supports the automatic generation of interconnection systems. This allows the WISHBONE interconnection to be generated with parametric core generators."

d) From the Glossary on p. 21: "Parametric Core Generator. A software tool used for the generation of IP cores based on input parameters. One example of a parametric core generator is a DSP filter generator. These are programs that create lowpass, bandpass and highpass DSP filters. The parameters for the filter are provided by the user, which causes the program to produce the digital filter as a VHDL or Verilog® hardware description. Parametric core generators can also be used create WISHBONE interconnections."

Item 6, <u>Altium Unveils new 'Board-on-Chip' Technology</u>, announces a technology known to those skilled in the art as being based upon WISHBONE (SoC) interconnection technology. With respect to the features of this tool listed on page 2 in this document:

a) "Mixed schematic/VHDL design capture"

b) "Processor core packs that combine pre-synthesised processor cores with matching compiler, simulator and debugger"

c) "Integrated software development"

d) "'Virtual' instruments such as logic analyzers and frequency counters that can be built into the design for test purposes"

Item 7, <u>Altium Introduces Systems Focus to FPGA Design with Nexar</u>, announces a technology known to those skilled in the art as: (i) being based upon WISHBONE (SoC) interconnection technology and (ii) further advancing the art of parametric core generators and test benches used in the product described in Item 6.

Respectfully submitted by:
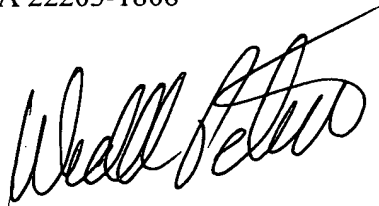
Wade D. Peterson

Certificate of Service

I hereby certify on this 22nd day of March, 2007 that a true and correct copy of the foregoing "Submission of Prior Art" was mailed by first-class mail, postage paid, to:

Nixon & Vanderhye, PC
John R. Lastova, Attorney at Law
901 North Glebe Road, 11th Floor
Arlington, VA USA 22203-1808
Tel: 703-816-4000

Signed:

Wade D. Peterson
Silicore Corporation
3500 Vicksburg Ln N. #110
Plymouth, MN 55447
TEL: 612-388-9755

THIS PAGE BLANK (USPTO)

VSO / July 21, 1999 / Vancouver, B.C., Canada

Emerging Technology...

System-on--chip (SOC) and its Effect on Microcomputer Bus Products

Wade D. Peterson, President & CEO
Silicore Corporation, Minneapolis, MN
TEL: (612) 722-3815 FAX: (612) 722-5841
peter299@maroon.tc.umn.edu   www.silicore.net

Silicore™

Electronic Design • Sensors • IP Cores

What We'll Cover Today

- Short introduction to system-on-chip & IP core technologies.
- The impact of SOC on the µC board industry.
- The need for new VMEbus interface chips.
- A sneak preview of new VMEbus IP cores.
- The WISHBONE interconnection (a µC bus for SOC).

2

What the Heck is System-on-chip (SOC)?

- System-on-chip is exactly what it says…the ability to build entire systems on a single semiconductor device.

- This includes CPU, RAM, ROM, I/O, FIFO, network interface, bus interface and all of the 'glue' logic.

- It's a lot like VMEbus or cPCI, except that the integration occurs at the chip level.

What's this 'IP Core' Thing Anyway?

- IP core: Intellectual Property Core.

- These are the building blocks for SOC.

- They contain the functional elements such as CPU, network interfaces, bus interface etc.

- There is currently a budding IP core industry.

- Many new IP cores are being released, almost daily.

4

So, What's all the Fuss About?

- SOC is a new paradigm in the semiconductor world.

- This is a juggernaut that can't be ignored!

- It alters the way we buy semiconductor technology.

- It's brought about by new chips and development tools.

- The full impact will be felt over the next 5-10 years.

Effects on the µC Board Industry

- SOC <u>will not</u> undermine the usefulness of µC boards:
  - There will always be a need for board level products.
  - End users won't want to screw around with SOC.

- SOC <u>will</u> change:
  - Several business aspects of µC boards.
  - How we design and deliver technology on µC boards.

Business Aspects

- SOC technology will allow µC board manufacturers to do a better job of differentiating their products.

- That's because SOC allows more choices for board builders.

- This will result in better profit margins for board manufacturers, and more choices for end users.

Technology Aspects

- Board makers and end users will also benefit, as it gives them a greater choice of technologies.

- Let's take a look at some of these choices...

Latest Semiconductor Technology

- We are starting to see a trend where the latest technologies are available first as IP cores.

- Great examples: NGIO, CPU and DSP cores.

- Companies will need to adopt SOC technologies if they want to maintain a technological 'edge'.

- Small, innovative companies are also creating IP Cores.

- That's because they're relatively cheap to develop.

The Big Ugly: Parts Obsolescence

- Parts obsolescence is a major concern in the industry.

- Soft IP cores are portable.

- As target devices (parts) become obsolete, they can be upgraded to the latest technology.

- This is really a return to the multiple vendor sources that we were accustomed to in the 1980's.

- This means less dependence on single-source vendors.

The Need to go Faster, Faster and still Faster

- SOC is an inherently high speed technology.
- On-chip capacitance and inductance is very low, especially when compared to circuit board technologies.
- FPGA toggle rates are now approaching 1 GHz.
- SOC is a very attractive way to increase system speed.

Other Benefits...

- SOC solutions require less power.

- SOC solutions are very compact.

- Parts are available in a wide variety of costs, speeds, packages, voltages and temperature ranges.

- Radiation hardened parts are available (military!).

- JTAG support is very common, and easy to do on FPGAs.

2

Drawbacks of SOC

- The industry is in the 'wild-west' stage.

- Some key, de-facto standards haven't developed yet.

- Tool costs are high, but are coming down.

- Lack of infrastructure among μC board manufacturers.

- Substantial learning curves.

- Lack of IP cores for VMEbus.

- Lack of an interconnection standard.

A Quick Overview of SOC integration

- Since many system-on-chip technologies are unfamiliar to the audience, let's take a look at how it's done...

14

SOC / System Integration Philosophy

- Microcomputer bus systems are integrated at the board level.

- SOC systems are integrated at the IP core level.

Target Devices

- SOC is used on FPGA and ASIC target devices.
- FPGA: Field Programmable Gate Array:
  - Lower entry cost, higher production cost (<10K pcs/yr).
  - Shorter lead times.
- ASIC: Application Specific Integrated Circuit:
  - Higher entry cost, lower production cost (>10K pcs/yr).
  - Longer lead times.

Integration of Soft IP Cores on an FPGA

Commercial IP Core

Commercial IP Core

User IP Core

User IP Core

User Logic

VHDL or Verilog Source

Integration

Vendor Specific I/O, RAM

Vendor Specific

Simulation

Typical Software Suppliers:

ModelSim
Synopsis
Synplicity
Accolade

Synthesis

Vendor Specific

Typical Software Suppliers:

Synopsis
Synplicity
Accolade

Place & Route

Vendor Specific

Supplied By FPGA Vendor

Completed Part

Vendor Specific

Typical Vendors:

Actel
Altera
Lucent
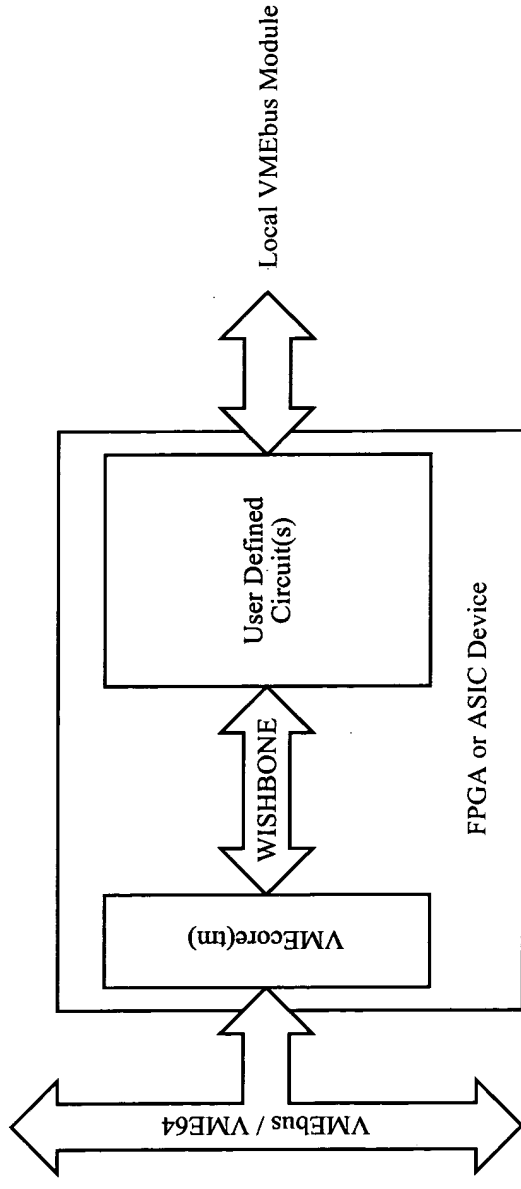Quicklogic
Xilinx

VHDL or Verilog Source

The Need For New VMEbus Semiconductors

- VMEbus interface chip technology is beginning to lag.
- Semiconductor manufacturers aren't keeping up anymore:
  - New interface standards (VME320, T&M, hot-swap, etc.).
  - Wide temperature and radiation hardening.
  - Packaging and voltages.
  - JTAG support (especially for ball grid array packaging).

18

Sneak Peak: VMEcore ™

Advance Information

- We decided to take advantage of this new situation by creating some new VMEbus interface chips.

- The new product is called VMEcore ™.

Local VMEbus Module

User Defined Circuit(s)

WISHBONE

VMEcore(tm)

FPGA or ASIC Device

VMEbus / VME64

VMEcore ™

Advance Information

- VMEcore™ is a VHDL Soft IP core toolkit.

- Suitable for FPGA or ASIC.

- Supports all interfaces in ANSI/VITA 1-1994 (VME64).

- Insures compliance with the VMEbus standard.

VMEcore ™

Advance Information

- Local WISHBONE interconnect.

- Full VMEbus bandwidth to 40/80 Mbyte/sec.

- Incorporates third generation synchronous technology.

- Clever design eliminates common interface problems.

21

VMEcore™
Advance Information

- The full toolkit will support:
  - Master
  - Slave
  - Location monitor
  - Interrupter
  - Handler
  - Bus arbiter

22

VMEcore ™

Advance Information

- A general purpose VMEbus IP core is very difficult.

- There are 10^{22} combinations of VMEbus interface!

- VMEcore ™ supports all of these combinations!

VMEcore ™

Advance Information

- A unique software package called the VMEbus Interface Writer™ creates a VHDL soft IP core to the user's specifications.

- VMEbus interface options are entered into the software, and it generates a VHDL soft IP core!

- A complete test bench (with test vectors) is also created!

- Here's a demonstration....

29

Standard Interconnection Architecture

- There is no standard interconnection architecture on SOC, at least nothing similar to a microcomputer bus.

- This makes it difficult to integrate systems.

- We have created the WISHBONE interconnection to solve this problem.

- WISHBONE is very similar to other µC buses.

- It's tailored toward SOC applications.

WISHBONE Architecture

- Simple, compact, logical IP core hardware interface.

- It's independent of:

  - Hardware technology (FPGA, ASIC, etc.).

  - IP core delivery method (soft, firm or hard).

  - Hardware description languages (VHDL, Verilog®, etc).

  - Synthesis, router or layout tools.

WISHBONE Specification

- Informal, open standard.

- No royalties or licensing required.

- The bus specification can be downloaded for free.

- See our website at www.silicore.net.

WISHBONE Features

- Synchronous protocol.

- One data transfer per clock cycle.

- Variable speed handshaking protocol.

- Speed is limited only by the target semiconductor technology.
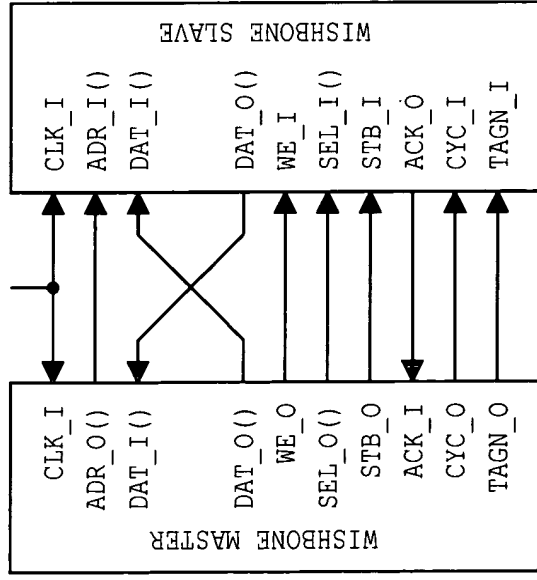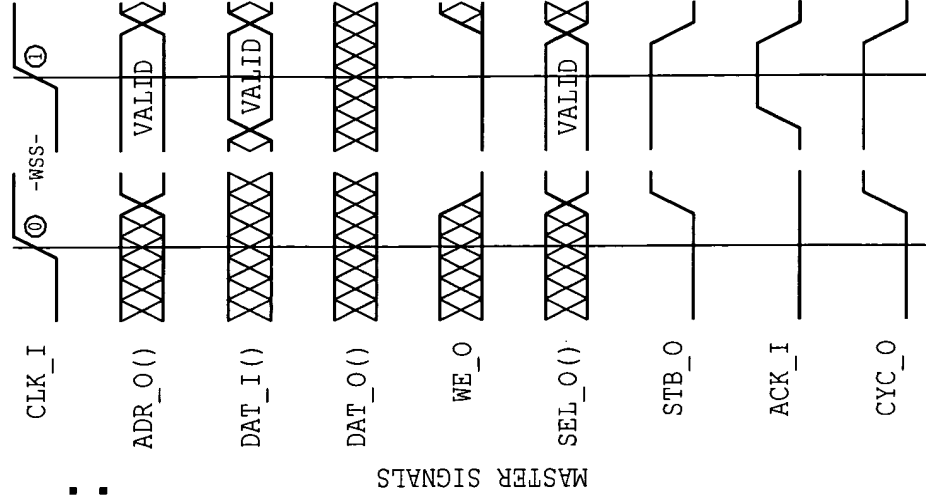
WISHBONE Features

- MASTER/SLAVE architecture.

- Up to 64-bit address and data paths (expandable).

- BIG ENDIAN or LITTLE ENDIAN byte lane ordering.

- Multiprocessing support.

- Memory-mapped, FIFO and crossbar configurations.

- Various interconnection methods (dual I/O, three-state).

WISHBONE Bus Cycles

- Full set of popular data transfer protocols, including:
  - READ/WRITE cycle
  - BLOCK transfer cycle
  - RMW cycle
  - EVENT cycle

30

Typical WISHBONE READ Cycle

- Here's a typical READ Cycle:

CLK_I

ADR_O()   VALID

DAT_I()   VALID

DAT_O()

WE_O

SEL_O()   VALID

STB_O

ACK_I

CYC_O

MASTER SIGNALS

-WSS-

WISHBONE SLAVE

CLK_I
ADR_I()
DAT_I()
DAT_O()
WE_I
SEL_I()
STB_I
ACK_O
CYC_I
TAGN_I

WISHBONE MASTER

CLK_I
ADR_O()
DAT_I()
DAT_O()
WE_O
SEL_O()
STB_O
ACK_I
CYC_O
TAGN_O
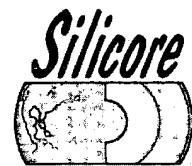
THIS PAGE BLANK (USPTO)

VME64 to PCI Bridge System-on-Chip (SoC)
Technical Reference Manual

Silicore Corporation

●

Silicore

Electronic Design
Sensors ● IP Cores

VME64 to PCI Bridge System-on-Chip (SoC)

Table of Contents

Manual Revision Level

Manual Revisions		
Manual Revision Level	**Date**	**Description of Changes**
0	02 JAN 2002	Preliminary release for comment. PCI Rev ID Code: 0x0000
1	30 SEP 2002	Project complete. PCI Rev ID Code: 0x0001
2	09 OCT 2002	Incorporate the following changes: - Fix typographical errors - Add SPECIALREG register - Add three-state clock enable to EEPROM clock (CEEPROM entity)
3	7 DEC 2002	Incorporate the following changes: - Reset all sections (except PCI core) after VMEbus [SYSRESET*]. - Add 'PCIResetMonitor' bit (D04) to the DMC_HW_CONTROL register. - PCI Rev ID Code: 0x0002
4	17 JAN 2004	Release under the GNU Free Documentation License
5	4 OCT 2004	Change of address (minor modification)

1.0 Introduction

The VME64 to PCI Bridge is a System-on-Chip that allows data transfer between a VMEbus slave interface and a PCI target interface. It is delivered as a VHDL soft core module that is intended for use on a Xilinx Spartan 2 FPGA. However, with minor modifications this soft system can be implemented on other types or brands of FPGA and ASIC devices. Figure 1-1 shows a functional diagram of the bridge.
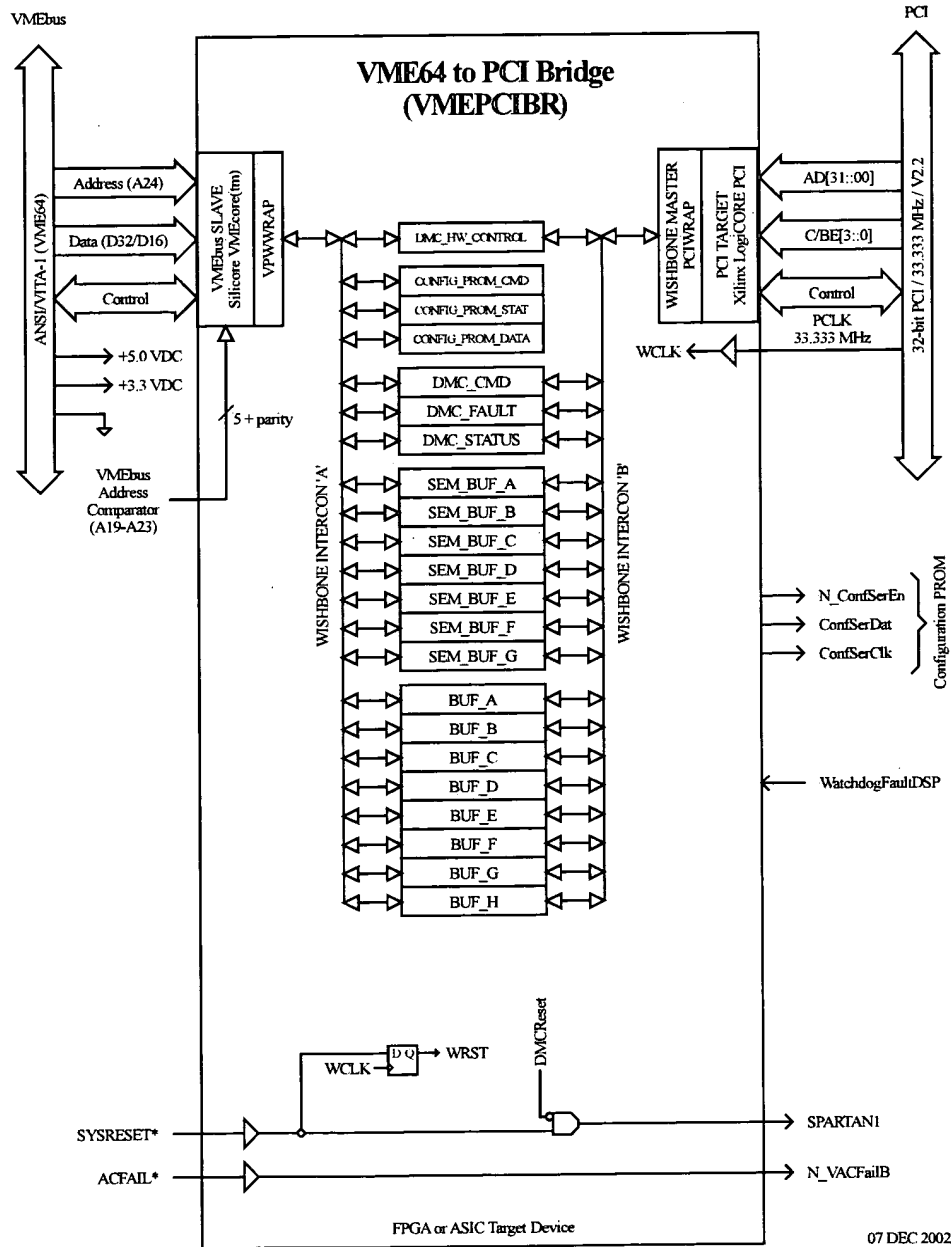
Figure 1-1. Functional diagram of the VMEbus to PCI bridge.

5

1.1 Features Of The Bridge

- VMEbus interface features:
    - Silicore VMEcore™ IP Core technology[1]
    - A24:D32:D16 slave interface
    - Posted read and write capabilities
    - Synchronous interface design
    - Conforms to ANSI/VITA 1 – 1994

- PCI interface features:
    - Xilinx LogiCORE™ PCI IP Core technology[2]
    - D32 target interface
    - 0 - 33.333 MHz operation
    - Conforms to PCI Revison 2.2

- Shared memory features:
    - Nine, 256 x 32-bit shared memory buffers
    - Semaphore control register for each buffer (except BUF H)
    - Buffers use independent memory arbitration

- Serial EEPROM interface for Atmel AT17 Series of FPGA Configuration ROM.

- Internal WISHBONE bus features[3]:
    - Conforms to WISHBONE Revision B.2
    - Simple, compact, logical hardware interfaces
    - Portable across FPGA and ASIC target devices
    - Third party support (including open source) is available at www.opencores.org

- Straightforward and highly reliable synchronous design.

- Written in flexible and portable VHDL hardware description language. All components (with the exception of the Xilinx LOGIcore PCI interface) are delivered as soft cores, meaning that all VHDL source code and test benches are supplied. This allows the end user to maintain and modify the design. Complete documentation is also provided.

---

[1] VMEcore™ is a VMEbus interface that is generated by the Silicore Bus Interface Writer™. The Bus Interface Writer is a parametric core generator that generates VHDL source code files.

[2] The VME64 to PCI Bridge SoC described in this manual interfaces to the back end of the Xilinx LOGIcore PCI, and is purchased separately from Xilinx. For more information, please refer to the Xilinx, Inc. web site at www.xilinx.com and the Xilinx LOGIcore PCI Design Guide.

[3] The internal WISHBONE SoC components were modified from source code in the public domain. Open source WISHBONE SoC components are available on the web from the WISHBONE Service Center at www.silicore.net/wishbone.htm or from www.opencores.org.

1.2 Glossary Of Terms

0x (numerical prefix)
The '0x' prefix indicates a hexadecimal number. It is the same nomenclature as commonly used in the 'C' programming language.

Active High Logic State
A logic state that is 'true' when the logic level is a binary '1' (high state). The high state is at a higher voltage than the low state.

Active Low Logic State
A logic state that is 'true' when the logic level is a binary '0' (low state). The low state is at a lower voltage than the high state.

ASIC
Acronym for: Application Specific Integrated Circuit. A general term which describes a generic array of logic gates or analog building blocks which are programmed by a metallization layer at a silicon foundry. High level circuit descriptions are impressed upon the logic gates or analog building blocks in the form of metal interconnects.

Asserted
(1) A verb indicating that a logic state has switched from the inactive to the active state. When active high logic is used it means that a signal has switched from a logic low level to a logic high level. (2) *Assert:* to cause a signal line to make a transition from its logically false (inactive) state to its logically true (active) state. Opposite of *negated*.

Bit
A single binary (base 2) digit.

Bridge
An interconnection system that allows data exchange between two or more buses. The buses may have similar or different electrical, mechanical and logical structures.

Bus Interface
An electronic circuit that drives or receives data or power from a bus.

Bus Cycle
The process whereby digital signals effect the transfer of data across a bus by means of an inter-locked sequence of control signals.

BYTE
A unit of data that is 8-bits wide. Also see: *WORD*, *DWORD* and *QWORD*.

Data Organization
The ordering of data during a transfer. Generally, 8-bit (byte) data can be stored with the most significant byte of a multi byte transfer at the higher or the lower address. These two methods

are generally called BIG ENDIAN and LITTLE ENDIAN, respectively. In general, BIG EN-DIAN refers to byte lane ordering where the most significant byte is stored at the lower address. LITTLE ENDIAN refers to byte lane ordering where the most significant byte is stored at the higher address. The terms BIG ENDIAN and LITTLE ENDIAN for data organization was coined by Danny Cohen of the Information Sciences Institute, and was derived from the book Gulliver's Travels by Jonathan Swift.

DWORD
A unit of data that is 32-bits wide. Also see: *BYTE, WORD* and *QWORD*.

ENDIAN
See the definition under 'Data Organization'.

Firm Core
An IP Core that is delivered in a way that allows conversion into an integrated circuit design, but does not allow the design to be easily reverse engineered. It is analogous to a binary or object file in the field of computer software design.

FPGA
Acronym for: Field Programmable Gate Array. Describes a generic array of logical gates and interconnect paths which are programmed by the end user. High level logic descriptions are im-pressed upon the gates and interconnect paths, often in the form of IP Cores.

Granularity
The smallest unit of data transfer that a port is capable of transferring. For example, a 32-bit port can be broken up into four 8-bit BYTE segments. In this case, the granularity of the interface is 8-bits. Also see: *port size* and *operand size*.

Hard Core
An IP Core that is delivered in the form of a mask set (i.e. a graphical description of the features and connections in an integrated circuit).

Hardware Description Language (HDL)
(1) Acronym for: Hardware Description Language. Examples include VHDL and Verilog®. (2) A general-purpose language used for the design of digital electronic systems.

IP Core
Acronym for: Intellectual Property Core. Also see: *soft core, firm core* and *hard core*.

Negated
A verb indicating that a logic state has switched from the active to the inactive state. When ac-tive high logic is used it means that a signal has switched from a logic high level to a logic low level. Also see: *asserted*.

Operand Size
The operand size is the largest single unit of data that is moved through an interface. For example, a 32-bit DWORD operand can be moved through an 16-bit port with two data transfers. Also see: *granularity* and *port size*.

PCI
Acronym for: Peripheral Component Interconnect. Generally used as an interconnection scheme (bus) between integrated circuits. It also exists as a board level interconnection known as Compact PCI (or cPCI).

Port Size
The width of a data port in bits. Also see: *granularity* and *operand size*.

Posted Read and Write Cycles
A method for minimizing data transfer latency between a data source and its destination. During a posted read or write cycle a bus interface, lying between a data source and its destination, captures the data and completes any handshaking protocols with the data source. At the same time, the bus interface initiates handshaking with the data destination. This alleviates the need for the data source to wait until the destination is ready to accept the data.

QWORD
A unit of data that is 64-bits wide. Also see: *BYTE, WORD* and *DWORD*.

Register (REG)
A device capable of retaining information for control purposes that is contained in a single BYTE, WORD or DWORD of storage area. Said storage area is not general purpose memory. Also see: *shared register (SREG)*.

Shared Memory (SMEM)
(1) The address space in a system which is accessible to all modules. (2) A type of memory that is shared between two or more ports. Shared memory uses a hardware arbitration scheme that allows simultaneous accesses from two or more processors. However, during simultaneous accesses one processor may be required to wait until the another has completed its accesses into the shared memory area. Also see: *shared register (SREG)*.

Shared Register (SREG)
(1) A register space in a system which is accessible to all modules. (2) A type of register that is shared between two or more ports. Shared registers uses a hardware arbitration scheme that allows simultaneous accesses from two or more processors. However, during simultaneous accesses one processor may be required to wait until the another has completed its accesses into the shared memory area. Also see: *register, shared memory (SMEM)*.

SoC
Acronym for System-on-Chip. Also see: *System-on-Chip*.

Soft Core
An IP Core that is delivered in the form of a hardware description language or schematic diagram.

System-on-Chip (SoC)
A method by which whole systems are created on a single integrated circuit chip. In many cases, this requires the use of IP cores which have been designed by multiple IP core providers. System-on-Chip is similar to traditional microcomputer bus systems whereby the individual components are designed, tested and built separately. The components are then integrated to form a finished system.

Target Device
The semiconductor type (or technology) onto which the IP core design is impressed. Typical examples include FPGA and ASIC target devices.

VHDL
Acronym for: VHSIC Hardware Description Language. [VHSIC: Very High Speed Integrated Circuit]. A textual based computer language intended for use in circuit design. The VHDL language is both a synthesis and a simulation tool. Early forms of the language emerged from US Dept. of Defense ARPA projects in the 1960's, and have since been greatly expanded and refined. Complete descriptions of the language can be found in the IEEE 1076, IEEE 1073.3, and IEEE 1164 specifications.

VMEbus
Acronym for: Versa Module Eurocard bus. A popular microcomputer (board) bus. Standardized under IEC 821, IEEE 1014 and ANSI/VITA 1-1994.

WISHBONE
A flexible System-on-Chip (SoC) design methodology. WISHBONE establishes common interface standards for data exchange between modules within an integrated circuit chip. Its purpose is to foster design reuse, portability and reliability of SoC designs. WISHBONE is a public domain standard.

Wrapper
A circuit element that converts a non-WISHBONE IP Core into a WISHBONE compatible IP Core. For example, consider a 16-byte synchronous memory primitive that is provided by an IC vendor. The memory primitive can be made into a WISHBONE compatible SLAVE by layering a circuit over the memory primitive, thereby creating a WISHBONE compatible SLAVE. A wrapper is analogous to a technique used to convert software written in 'C' to that written in 'C++'.

WORD
A unit of data that is 16-bits wide. Also see: *BYTE, DWORD* and *QWORD.*

1.3 Recommended Skill Level

It is recommended that the user have some experience with VHDL syntax and synthesis before attempting to integrate this core (or almost any other HDL core for that matter) into an FPGA or ASIC device. Most VHDL users report a fairly stiff learning curve on their first project, so it's better to have that experience before attempting to integrate the core. Prior experience with one or two medium size VHDL projects should be sufficient. On the other hand, some users may find the integration of the core a good way to learn many of the concepts in the VHDL language. Those users should find the integration experience rewarding.

1.4 References

- Ashenden, Peter J. The Designer's Guide to VHDL. Morgan Kaufmann Publishers, Inc. 1996. ISBN 1-55860-270-4. Excellent general purpose reference guide to VHDL. Weak on synthesis, stronger on test benches. Good general purpose guide, very complete.

- IEEE Standard VHDL Language Reference Manual. IEEE Std 1076-1993. IEEE, New York NY USA 1993. This is a standard, and not a tutorial by any means. Useful for defining portable VHDL code.

- IEEE Standard VHDL Synthesis Packages. IEEE Std 1076.3-1997. IEEE, New York NY USA 1997. This is a standard, and not a tutorial by any means. Useful for defining portable VHDL code.

- IEEE Standard Multivalue Logic System for VHDL Model Interoperability (Std logic 1164). IEEE Std 1164-1993. IEEE, New York NY USA 1993. This is a standard, and not a tutorial by any means. Useful for defining portable VHDL code.

- Pellerin, David and Douglas Taylor. VHDL Made Easy. Prentice Hall PTR 1997. ISBN 0-13-650763-8. Good introduction to VHDL synthesis and test benches, and closely follows the IEEE standards.

- PCI Special Interest Group. PCI Local Bus Specification Revision 2.0.

- Peterson, Wade D. The VMEbus Handbook, 4th Ed. VITA 1997 ISBN 1-885731-08-6

- Peterson, Wade D. WISHBONE System-on-Chip (SoC) Interconnection Architecture for Portable IP Cores. Revision B.2.

- Schmitz, Manfred. "PCI-to-VME Bridging: Keeping the Best of Both Worlds". RTC Magazine, Dec 2001. pp 59-64.

- Shanley, Tom and Don Anderson. PCI System Architecture 4th Ed. Addison-Wesley 2001. ISBN 0-201-30974-2.

- Skahill, Kevin. VHDL For Programmable Logic. Addison-Wesley 1996. ISBN 0-201-89573-0. Excellent reference for VHDL synthesis. Very good treatment of practical VHDL code for the synthesis of logic elements. Weak on test benches and execution of the IEEE standards.

- VITA. American National Standard for VME64: ANSI/VITA 1-1994.

- Xilinx, Inc. LogiCORE™ PCI Design Guide Version 3.0. Xilinx Inc 2001.

2.0 System Architecture

The VME64 to PCI Bridge connects a VMEbus slave interface to a PCI target interface. Communication between the two sides of the bridge is made through a set of four control registers, seven semaphore registers and nine shared memory buffers. Data is loaded into a buffer on one side of the bridge, and unloaded from the other side. The semaphore registers can also be used by system software to determine when and if a buffer is being used. The VMEbus side of the bridge also includes an EEPROM interface for programming Atmel AT17 series devices. Table 2-1 shows the address map from both sides of the bridge.

VMEbus BYTE Address[4] (Base Offset)	PCI BYTE Address (BAR Offset)	Name	Type	Access Types
0x00000	0x0000	DMC_HW_CONTROL	SREG	R/W
0x00004	(*)	CONFIG_PROM_CMD	REG	R/W
0x00008	(*)	CONFIG_WRITE_DATA	REG	R/W
0x0000C	(*)	CONFIG_READ_DATA	REG	R
0x00010	0x0010	DMC_CMD	SREG	R/W
0x00014	0x0014	DMC_FAULT	SREG	R/W
0x00018	0x0018	DMC_STATUS	SREG	R/W
0x0001C	0x001C	SPECIALREG	SREG	R/W
0x00020	0x0020	SEM_BUF_A	SREG	R/W
0x00024	0x0024	SEM_BUF_B	SREG	R/W
0x00028	0x0028	SEM_BUF_C	SREG	R/W
0x0002C	0x002C	SEM_BUF_D	SREG	R/W
0x00030	0x0030	SEM_BUF_E	SREG	R/W
0x00034	0x0034	SEM_BUF_F	SREG	R/W
0x00038	0x0038	SEM_BUF_G	SREG	R/W
0x0003C – 0x007FF	0x003C – 0x07FF	Unused / Unreserved	-	-
0x00800 – 0x00BFF	0x0800 – 0x0BFF	BUF_A	SMEM	R/W
0x00C00 – 0x00FFF	0xC00 – 0x0FFF	BUF_B	SMEM	R/W
0x01000 – 0x013FF	0x1000 – 0x13FF	BUF_C	SMEM	R/W
0x01400 – 0x017FF	0x1400 – 0x17FF	BUF_D	SMEM	R/W
0x01800 – 0x01BFF	0x1800 – 0x1BFF	BUF_E	SMEM	R/W
0x01C00 – 0x01FFF	0x1C00 – 0x1FFF	BUF_F	SMEM	R/W
0x02000 – 0x023FF	0x2000 – 0x23FF	BUF_G	SMEM	R/W
0x02400 – 0x027FF	0x2400 – 0x27FF	BUF_H	SMEM (**)	R/W
0x02800 – 0x0FFFF	0x2800 – 0xFFFF	Unused / Unreserved	-	-
0x10000 – 0x7FFFF	-	Unused / Unreserved	-	-

Table 2-1. Address Map.

Notes:
Access types: 'R': read cycle, 'W': write cycle.
Type: REG (register), SREG (shared register) and SMEM (shared memory) are defined in the glossary.
(*) Not implemented on the PCI side of the bridge.
(**) 'BUF_H' is formed from Xilinx distributed RAM, and is the only region that will accept PCI burst transfers. All other buffers will only respond to single read and write cycles.

---

[4] By definition, VMEbus and PCI addresses are specified at byte locations. The first address given in the table is the byte address that should be used when accessing the register or memory area using DWORD (32-bit) values.

2.1 VMEbus Interface

The VMEbus slave interface is compatible with ANSI/VITA 1-1994. It has a D32:D16 data interface, and responds to A24 SLAVE base addresses[5] on 512 Kbyte boundaries. Stated another way, the interface responds to A24 base address locations starting at byte locations of 0x000000, 0x080000, 0x100000 and so forth. For example, if the board resides at base address 0x0C0000, then the 'DMC_CMD' register can be accessed at 0x100000 + 0x000010 = 0x100010.

The bridge responds to VMEbus single read and write cycles. It does not respond to the VMEbus block transfer cycle, nor does it support the VMEbus read-modify-write (RMW) cycle.

All registers and buffer memory areas have port sizes of 32-bits (DWORD). All have 16-bit (WORD) granularity, meaning that they can be accessed as 16 or 32-bit quantities. Byte accesses are not supported.

The VMEbus interface is almost, but not totally compliant, with ANSI/VITA 1-1994. That specification requires (under RULE 2.77) that the interface support byte, or D08(EO), data transfer modes. However, this interface only responds to WORD (16-bit) accesses on 2-byte boundaries, and DWORD (32-bit) accesses on 4-byte boundaries[6].

All VMEbus accesses to 'unused' address areas are terminated with the VMEbus [BERR*] signal.

2.2 PCI Interface

The PCI target interface is implemented with a Xilinx LogiCORE PCI interface[7]. Unless otherwise noted, it is compatible with the PCI Version 2.2 bus specification.

The target interface responds to the following PCI command cycles:

- Configuration Read (CBE[3:0] = 1010)
- Configuration Write (CBE[3:0] = 1011)
- Memory Read (CBE[3:0] = 0110)
- Memory Write (CBE[3:0] = 0111)
- Memory Read Multiple (CBE[3:0] = 1100)
- Memory Read Line (CBE[3:0] = 1110)

The PCI interface responds to 64 Kbyte of memory I/O space starting at the address programmed in the BAR0 register.

---

[5] The VMEbus interface responds to A24 address modifier codes 0x3E, 0x3D, 0x3A and 0x39.
[6] This caveat is required because the target device for the bridge core is the Xilinx Spartan 2 FPGA. The limited number of block memories on that device forces an internal memory architecture (granularity) of 16-bits.
[7] For more details, please refer to the Xilinx LogiCORE PCI Design Guide.

All registers and buffer memory areas located on the PCI target interface have port sizes of 32-bits (DWORD). All have 16-bit (WORD) granularity, meaning that they can be accessed as 16 or 32-bit quantities. The bridge uses a 16/32-bit data interface, which means that it can read and write 16-bit WORD or 32-bit DWORD values. The interface does not support BYTE accesses.

All PCI accesses to 'unused' address areas are terminated with a PCI TARGET ABORT.

Special 'wrapper' circuitry connected to Xilinx LogiCore PCI interface controls how the target interface responds to the PCI commands. During the initial phase of a PCI burst cycle, a binary counter (located inside the wrapper circuitry) latches the starting PCI address. The counter then increments during subsequent phases within the burst cycle, thereby generating the next address. The counter is incremented after every cycle that accesses the high byte of a 32-bit DWORD transfer. The high byte is indicated when [S_CBE(3)] is low. That means that the address counter is incremented after every 32-bit transfer or after a high order 16-bit transfer.

PCI single and burst transactions are supported by the bridge. However, burst transactions[8] are not supported by all memory locations or registers. If a burst transaction is attempted in a memory region that does not support burst transfers, then the bridge responds with a TARGET ABORT termination[9].

Reading or writing to more than one 'SREG' or 'SMEM' area during a single PCI burst cycle may result in an unexpected behavior, and is not recommended.

Also note that the 8-bit address counter rolls over from 0xFF to 0x00 during burst cycles. This means that burst transfers cannot cross boundaries from one shared memory to another. For example, a 256 DWORD burst to the middle of 'BUF_H' will wrap around to the beginning of the same buffer.

The Xilinx LogiCORE PCI handles all transactions with the bridge. The core is implemented as a PCI target, meaning that it cannot initiate transactions. This manual does not attempt to define the operation of the Xilinx LogiCORE PCI interface.

The PCI core returns a PCI Device ID and a PCI Rev ID that are unique to the VMEbus to PCI Bridge core. These are returned from PCI configuration registers 0x00 and 0x08 respectively. The PCI Device ID always returns 0x030. The PCI Rev ID identifies the hardware revision level as shown in the 'Manual Revision' section at the front of this manual.

---

[8] Burst transactions are bus cycles with more than one data transfer phase.
[9] Burst transactions in excess of one data transfer are allowed as long as the memory structure supports it. This is because the core can be implemented on a number of target devices, memory types and speeds. The Xilinx Logi-Core PCI requires that memories must support single clock data transfers. If this type of memory is implemented on the device, then the burst operation is supported. If this type of memory is not implemented with the core, then burst transactions are not supported. For more information please refer the Hardware Reference section of this manual.

2.3 Register Descriptions

Unless otherwise noted, the VMEbus and PCI shared registers (SREG) have identical bit descriptions.

2.3.1 DMC_HW_CONTROL Register

Tables 2-2 and 2-3 show the operation of the DMC_HW_CONTROL Register.

Table 2-2. DMC_HW_CONTROL Definition			
Bit #	Name	Write	Read
D00	DMCReset	0 = Local run (*) 1 = Local reset	Readback
D01	SysfailLocalDriver	0 = Assert SYSFAIL (*) 1 = Negate SYSFAIL	Readback
D02	SysfailSystemMonitor	0	0 = SYSFAIL asserted 1 = SYSFAIL negated
D03	AcfailSystemMonitor	0	0 = ACFAIL asserted 1 = ACFAIL negated
D04	PCIResetMonitor	0	0 = No PCI reset 1 = PCI reset
D05-D31	Unused / Unreserved	0	0

Notes:
(1) Always set unused bits to '0' to support future upgrades.
(2) Condition after VMEbus reset or device configuration denoted by: (*).

16

Table 2-3. DMC_HW_CONTROL Detailed Description

Bit #	Detailed Description
D00 DMCReset	WRITE: Clearing this bit asserts the [SPARTAN1] output pin on the device. Setting this bit negates the [SPARTAN1] output pin. [SPARTAN1] is an active high signal. READ: Returns the current state of the WRITE bit.
D01 SysfailLocalDriver	WRITE: Clearing this bit causes the local SYSFAIL* driver to assert the VMEbus SYSFAIL* signal. Setting this bit negates the local SYSFAIL driver. For more information about the operation of this bit, please see the *SYSFAIL* Operation* section of this manual. READ: Returns the current state of the WRITE bit.
D02 SysfailSystemMonitor	WRITE: Always set to '0' to support future upgrades of this register. READ: Returns the current state of the VMEbus SYSFAIL* signal. When cleared, the VMEbus SYSFAIL* signal is asserted (i.e. failure mode). When negated, the VMEbus SYSFAIL signal is negated.
D03 AcfailSystemMonitor	WRITE: Always set to '0' to support future upgrades of this register. READ: Returns the current state of the VMEbus ACFAIL* signal. When cleared, the VMEbus ACFAIL* signal is asserted (i.e. AC power failure mode). When negated, the VMEbus ACFAIL* signal is negated.
D04 PCIResetMonitor	WRITE: Always set to '0' to support future upgrades of this register. READ: Returns the current state of the PCI [RST#] signal. When set, the PCI bus is in its reset condition. When cleared, PCI bus is in its 'run' condition.
D05-D31 Unused / Unreserved	WRITE: Always set to '0' to support future upgrades of this register. READ: Always returns '0'.

The VME64 to PCI Bridge supports a standard implementation of the SYSFAIL* line. This includes both its start-up diagnostic and run-time failure capabilities. While these capabilities are not defined by the VMEbus specification, they do conform to standard industry practices.

SYSFAIL* is an open-collector class VMEbus signal. That means it operates as a 'wire-nor' circuit that is asserted if one or more VMEbus modules drives it low. It is negated only after all VMEbus modules negate their on-board SYSFAIL* drivers.

The standard start-up diagnostic operation of SYSFAIL* is shown in Figure 2-1. Under that practice, SYSFAIL* is asserted by all modules in response to the VMEbus SYSRESET* signal. After SYSRESET* is negated, all modules are tested to see if they are operating correctly. As each module passes its test it negates its SYSFAIL* driver. SYSFAIL* is negated only after all modules pass these diagnostic tests. This procedure follows a standard industry practice where all modules boot up in a failed state (i.e. they are assumed to be 'bad' until proven 'good'). SYSFAIL* is negated only after all of the modules are proven to be good.


```
SYSRESET*                                    Self test performed
                                             on all modules.
                              SYSFAIL* asserted
                              by on all modules
SYSFAIL*                      after SYSRESET*.

                                                      SYSFAIL* negated
                                                      after self test
                                                      passes on all modules.
                                                      19 SEP 2002
```
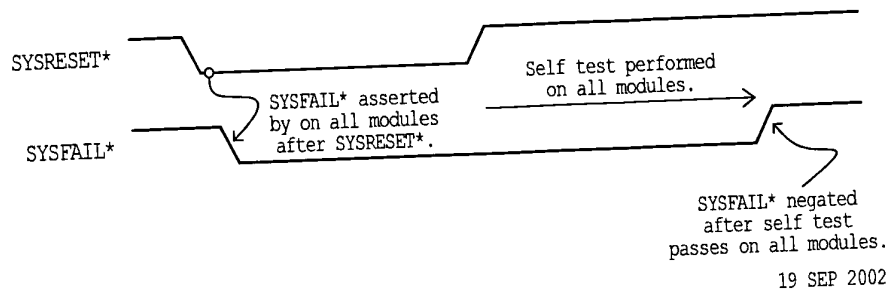
Figure 2-1. Standard VMEbus SYSFAIL* operation.

Very often a red/green LED is attached to a module's local SYSFAIL* driver. This allows quick diagnosis of the system at boot up time by an operator. Under this situation the system is booted and all LEDs turn red. As each module passes its diagnostic test, its LED turns green. In this way the system operator can quickly determine if one or more modules has failed.

A run-time failure can be indicated by a module at any time. To indicate the failure, the module simply asserts SYSFAIL*.

The VME64 to PCI Bridge has several capabilities that are specifically designed to support SYSFAIL* operation. Control and status bits are available from the DMC_HW_CONTROL VMEbus register.

The SYSFAIL* control and status bits are configured so that they look like an emergency stop (ESTOP) button. ESTOP buttons are a common practice in industrial systems. These are generally arranged into a 'wire-or' chain whereby pushing any ESTOP button in the chain causes the system to shut down in a known and controllable manner. This is a standard technique in all systems where a failure could cause damage to life or property. Commonly, ESTOP is a large red button that is pushed to immediately stop all motion or other hazardous conditions (such as de-energizing high voltage power supplies).

A common practice in industrial control systems is to provide an electronic ESTOP button for the control system. This allows the control system to safely power down the system in response to a failure. A common practice in VMEbus systems is to provide each critical module with it's own ESTOP button, and to wire-or them together with the SYSFAIL* line. SYSFAIL* on the backplane can then be wired to a relay (or other logic) that connects into the main system ESTOP chain.

18

The VME64 to PCI Bridge provides this capability on both sides of the bridge. Furthermore, each side of the bridge can monitor the other side of the bridge. For more information, please refer to the register descriptions.

2.3.2 Configuration EEPROM Registers

The Configuration EEPROM Interface assists in programming of the Atmel AT17 Series of FPGA Configuration EEPROM. The interface is configured by three registers:

- o   CONFIG_PROM_CMD: Command/status register
- o   CONFIG_WRITE_DATA: Write data register (8-bits).
- o   CONFIG_READ_DATA: Read data register (8-bits).

The user is encouraged to study the following data sheets (provided by Atmel) before programming a device:

1) *Programming Specification for Atmel's AT17 and AT17A Series FPGA Configuration EEPROMS.* Atmel Corporation 2002. See www.atmel.com.

2) *AT17LV040 FPGA Configuration EEPROM Memory* (data sheet).

The CONFIG_PROM_CMD register is shown in Tables 2-4 and 2-5. The bits in this register control the various operations of the interface. The CONFIG_READ_DATA and CONFIG_WRITE_DATA registers are for reading and writing data to the EEPROM. Load the lower eight bits of the CONFIG_WRITE_DATA register before issuing a write command, and read the lower eight bits of the CONFIG_READ_DATA register after completion of a read command.

The EEPROM interface can be reset (initialized) in one of two ways: (1) under hardware control in response to a system reset (via the WISHBONE reset line [RST_I]) or (2) under software control in response to the 'ResetPromInterface' bit in the CONFIG_PROM_CMD register.

After a system reset the 'ResetPromInterface' bit is set by hardware. This causes all sections of the EEPROM interface to be initialized. The reset operation may take some time to complete, as the interface operates with a clock speed of about 400 KHz. This means that the reset interval can last in excess of 3 – 6 microseconds. During this time all processor inquires to the 'ResetPromInterface' will return a 'BUSY' state. While busy, the host processor should not attempt to initiate EEPROM accesses.

To reset the EEPROM interface, the VMEbus host processor should perform the following:

1) Reset the interface by setting the 'ResetPromInterface' bit (D00) in the CONFIG_PROM_CMD register.
2) Monitor bit D00 ('BUSY') until the bit is cleared.
3) Perform other operations as needed.

Bit #	Name	Write	Read
D00	ResetPromInterface	0 = Operate interface (*) 1 = Reset interface	0 = Not busy 1 = Busy
D01	SendStartCondition	0 = No operation (*) 1 = Send start condition	0 = Not busy 1 = Busy
D02	SendStopCondition	0 = No operation (*) 1 = Send stop condition	0 = Not busy 1 = Busy
D03	WriteByteWithAck	0 = No operation (*) 1 = Write data byte	0 = Not busy 1 = Busy
D04	ReadByteWithAck	0 = No operation (*) 1 = Read data byte	0 = Not busy 1 = Busy
D05	ReadByteWithStop	0 = No operation (*) 1 = Read data byte	0 = Not busy 1 = Busy
D06	AckStat	0	0 = ACK received 1 = No ACK received
D07	ConfSerEn	0 = Disable configuration (*) 1 = Enable configuration	Readback
D08	ManualConfigOverride (Manual Override)	0 = Normal operation (*) 1 = Manual configuration	Readback
D09	ConfSerClkOverride (Manual Override)	0 = ConfSerClk cleared (*) 1 = ConfSerClk set	Readback
D10	ConfSerDatOverride (Manual Override)	0 = ConfSerDat cleared (*) 1 = ConfSerDat set	Readback
D11	ConfSerDat (Manual Override)	0	Input State
D12-D31	Unused / Unreserved	0	0

Table 2-4. CONFIG_PROM_CMD

Notes:
(1) Always set unused bits to '0' to support future upgrades.
(2) Condition after system reset or device configuration denoted by: (*).

Table 2-5. CONFIG_PROM_CMD Detailed Description

Bit #	Detailed Description
D00 ResetPromInterface	WRITE: Setting this bit resets the EEPROM interface. Negating the bit has no effect. This bit need only be set once to initiate a reset. READ: Returns the current state of the reset that was initiated by asserting the WRITE bit. The bit returns '1' while the interface is busy implementing the reset operation, and '0' when it has completed. NOTE: a 'ResetPromInterface' command is initiated in response to a system reset or device configuration. That means that host processor could read this bit as a '1' shortly after a system reset. However, the bit will be eventually be negated when the EEPROM interface has completed its reset cycle.
D01 SendStartCondition	WRITE: Setting this bit initiates an EEPROM START CONDITION. Negating the bit has no effect. READ: Returns the current state of the START CONDITION which was initiated by asserting the WRITE bit. The bit returns '1' while the interface is busy implementing the START CONDITION, and '0' when it has completed.
D02 SendStopCondition	WRITE: Setting this bit initiates an EEPROM STOP CONDITION. Negating the bit has no effect. READ: Returns the current state of the STOP CONDITION which was initiated by asserting the WRITE bit. The bit returns '1' while the interface is busy implementing the STOP CONDITION, and '0' when it has completed.
D03 WriteByteWithAck	WRITE: Setting this bit writes a byte of data to the EEPROM. Negating the bit has no effect. Before setting this bit the data must be loaded into lower eight bits of the CONFIG_WRITE_DATA register. READ: Returns the current state of the write operation which was initiated by asserting the WRITE bit. The bit returns '1' while the interface is busy writing data, and '0' when it has completed. The state of the 'ACK' bit (returned by the EEPROM) is indicated by bit D06 (below).

Table 2-5. CONFIG_PROM_CMD Detailed Description (con't)

Bit #	Detailed Description
D04 ReadByteWithAck	WRITE: Setting this bit reads a byte of data from the EEPROM and terminates it with an ACK CONDITION. Negating the bit has no effect. READ: Returns the current state of the read operation which was initiated by asserting the WRITE bit. The bit returns '1' while the interface is busy writing data, and '0' when it has completed. Note: After completion of the 'ReadByteWithAck' command, the EEPROM data byte is returned in the lower eight bits of the CONFIG_READ_DATA register.
D05 ReadByteWithStop	WRITE: Setting this bit reads a byte of data from the EEPROM and terminates it with a STOP CONDITION. Negating the bit has no effect. The data is returned in the CONFIG_READ_DATA register. READ: Returns the current state of the read operation which was initiated by asserting the WRITE bit. The bit returns '1' while the interface is busy writing data, and '0' when it has completed. Note: After completion of the 'ReadByteWithStop' command, the EEPROM data byte is returned in the lower eight bits of the CONFIG_READ_DATA register.
D06 AckStat	WRITE: Always write a '0' to this location. READ: Returns the current state of the ACK bit after implementing the WriteByteWithAck and ReadByteWithAck commands. Query this bit to determine if the EEPROM successfully acknowledged the commands.
D07 ConfSerEn	WRITE: Setting this bit enables configuration of the EEPROM. Negating the bit disables configuration. Always set this bit before attempting to read or write to the EEPROM. Always negate it when completed. READ: Returns the state of the WRITE bit.

Bit #	Detailed Description
Table 2-5. CONFIG_PROM_CMD Detailed Description (con't)	
D08 ManualConfigOverride	WRITE: Clearing this bit allows the interface to operate normally. Setting this bit overrides all other functions and allows the ConfSerClk and ConfSerDat to be operated manually. This function is used for test and debugging purposes. READ: Returns the state of the WRITE bit.
D09 ConfSerClkOverride	WRITE: asserts or negates the ConfSerClk signal when the ManualConfigOverride bit is set. READ: Returns the state of the WRITE bit.
D10 ConfSerDatOverride	WRITE: asserts or negates the ConfSerDat signal when the ManualConfigOverride bit is set. READ: Returns the state of the ConfSerDat signal.
D11 ConfSerDat Input	WRITE: Always write a '0' to this location. READ: Returns the state of the ConfSerDat input.
D12 – D31 Unused / Unreserved	WRITE: Always write a '0' to this location. READ: Always returns '0'.

When programming a configuration EEPROM the user is directed to the programming algorithm in the Atmel Programming specification. The interface provides much of the timing. For example, Figure 2-2 shows how the interface implements the first few bytes of data shown in the "Write to Whole Device" algorithm.

The Atmel AT17 series parts must be programmed at a speed that is sufficiently high to guarantee the 'Write Cycle Time' (T_{WR}) indicated in the data sheet. Stated another way, data written to the device must be delivered within a minimum length of time. For example, the Write Cycle Time of the Atmel AT17LV040 is specified as 25.0 ms (max). This means that the theoretical, maximum time required to deliver the data to the EEPROM is:

256 data bytes/page x 9 bits/byte[10] = 2,304 data bits/page

4 address bytes/page x 9 bits/byte = 36 address bits/page

2304 data bits/page + 36 address bits/page = 2,340 bits/page

---

[10] An address or data byte includes eight bits of information plus one acknowledge bit, or 9 bits total.

The EEPROM interface[11] operates at an clock speed (ECLK) of 393 KHz, and each address or data bit requires 4 clock cycles to complete. This means that the minimum, total Write Cycle Time is:

$$2{,}340 \text{ bits/page} \times 4 / 393 \text{ KHz} = 23.8 \text{ ms}$$

Under best case conditions a block of data takes at least 23.8 ms to write to the EEPROM. However, since the maximum Write Cycle Time for the AT17LV040 is 25.0 ms, this leaves only a very small guard band (1.2 ms) to meet the specification.

In order to achieve this data rate the host processor must deliver data to the interface in an expedient manner. During write cycles this usually means that back-to-back data write cycles (using the 'WriteByteWithAck' command) must occur seamlessly...one after the other. Stated another way, a new 'WriteByteWithAck' command must be issued immediately after the 'BUSY' bit has been negated from a previous command.

Once the EEPROM interface hardware has written a data byte to the device with the 'WriteByteWithAck' command, it negates the associated 'BUSY' bit. Once this happens, the host processor has [Tav] seconds to load the CONFIG_WRITE_DATA register and set the 'WriteByteWithAck' command bit. If the host processor does not meet [Tav] the interface will probably work just fine. However, if [Tav] is exceeded for too many cycles, then there is a risk that maximum Write Cycle Time of 25.0 ms could be exceeded.

The maximum time available [Tav] from 'BUSY' negated to 'WriteByteWithAck' set is found from the relation:

$$Tav = \frac{1}{ECLK} - \frac{6}{CLK\_I} - ECLKsetup$$

Given an EEPROM clock [ECLK] of 393 KHz, a WISHBONE clock speed [CLK_I] of 33.0 MHz and [ECLKsetup] of 1 WISHBONE clock yields a [Tav] of:

$$Tav = \frac{1}{393KHz} - \frac{6}{33MHz} - \frac{1}{33MHz} = 2.34\ uS$$

The system integrator / software programmer should also remember that non real-time operating systems (such as Unix or Windows®) may introduce significant time delays into software execution. These delays can be large when compared to the maximum Write Cycle Time of 25.0 ms. Use of such non deterministic software systems should be carefully evaluated.

Important Notice:
The first byte on the AT17 series EEPROM cannot be reliably read by the FPGA unless power is cycled after programming. For more information, please refer to the programming instructions.

---

[11] For more information about the EEPROM interface please refer to the 'CEEPROM.VHD' hardware reference located elsewhere in this manual.

START

ResetPromInterface = 1

ResetPromInterface 'Busy' (READ bit) ? — YES

NO

SendStartCondition = 1;

SendStartCondition 'Busy' (READ bit) ? — YES

NO

Load Device Address (AE\h or A5\h) Into CONFIG_WRITE_DATA

WriteByteWithAck = 1

WriteByteWithAck 'Busy' (READ bit) ? — YES

NO

'ACK Received' (READ bit DO6) ? — NO

YES

Load MSB of EEPROM address into CONFIG_WRITE_DATA

WriteByteWithAck = 1

WriteByteWithAck 'Busy' (READ bit) ? — YES

NO

'ACK Received' (READ bit DO6) ? — NO

YES

...etc.

EEPROM High Level Software Algorithm
17 AUG 2002

Figure 2-2. High level software control algorithm for the EEPROM Interface.

2.3.5 DMC_CMD Register

DMC_CMD is a 32-bit, user defined, general purpose read/write register.

2.3.6 DMC_FAULT Register

DMC_FAULT is a 32-bit read/write register, with the bit definitions shown in Table 2-6.

Table 2-6. DMC_FAULT Definition			
Bit #	Name	Write	Read
D00	WatchdogFaultDSP	0	'0' = No timeout '1' = Timeout
D01-D31	User defined	Latched	Returns current state

2.3.7 DMC_STATUS Register

DMC_STATUS is a 32-bit, user defined, general purpose read/write register.

2.3.8 SPECIALREG Register

SPECIALREG is a 32-bit, user defined, general purpose read/write register.

2.3.9 PCI_SEM_BUF_(A-G)

The seven semaphore registers SEM_BUF_(A-G) are used by VMEbus or PCI system processors to obtain ownership rights for the seven buffer memories. Table 2-7 is representative of the seven semaphore registers. Each SEM_BUF(A-G) semaphore register corresponds to a related BUF(A-G) shared buffer memory.

Table 2-7. SEM_BUF_(A-G) Definition			
Bit #	Name	Write	Read
D00	SemaphoreBufferN $(G \geq N \geq A)$	0 = Buffer released 1 = No effect	0 = Grant 1 = Busy
D01-D31	Unused / Unreserved	0	0

The 'SemaphoreBufferN' bit indicates whether a VMEbus or PCI system processor has obtained the memory space. The semaphore does not lock the memory buffer but just provides a mechanism for software to determine if the particular memory buffer is being used. If the semaphore is not used, or is disregarded by software, the associated buffer arbitration still operates normally.

The semaphore is accessed by reading the bit. If the bit is returned as '0', then the processor has obtained ownership of the buffer. If the bit is returned as '1', the buffer is busy. If a processor obtains the buffer by reading '0' (becomes the owner), and the bit is sampled for a second time, then the bit is returned as '1' on the second access.

If a VMEbus and PCI semaphore access occurs at the same time (i.e. during the same clock cycle), then the VMEbus access has priority.

The buffer is released by writing a '0' to the semaphore. Writing a '1' to the bit does not have any effect. The buffer may be released from either the VMEbus or PCI side of the bridge.

2.4 Operation of Shared Memory Buffers and Registers

The VME64 to PCI Bridge contains several different types of memories and buffers. These are classified as SMEM (shared memory), REG (register) and SREG (shared register). The classification for any particular memory or register can be found in the VMEbus and PCI address maps located elsewhere in this manual.

SMEM and SREG types are shared by both sides of the bridge[12]. Each has its own arbiter circuit that resolves any contention between the two sides of the shared resource. For the most part, this operation is transparent to the software programmer. However, in some cases it is important to understand how the arbitration works.

2.4.1 Shared Buffers

There are eight shared memory buffers named 'BUF_A' through 'BUF_H'. These can be used to pass data between the two sides of the bridge. They operate as 32-bit wide memories with 16-bit granularity. This means that information can be passed in WORD (16-bit) and DWORD (32-bit) data formats.

Each buffer operates as an independent shared memory. This means that both sides of the bridge supports full, simultaneous, read/write privileges into each buffer. This provides the software designer with a very flexible mechanism for moving data, as well as an excellent way to test memory from both sides of the bridge.

There may be some undesirable side effects as a result of contention within the shared areas. Specifically, the interface may slow down because a hardware arbiter must grant accesses to one

---

[12] A third type called 'REG' (register) is a non-shared register. These operate independently from the opposite side of the bridge.

27

side of the bridge or the other. To alleviate this problem, multiple shared memory buffers are provided. This allows one side of the bridge to access one buffer while the other side of the bridge accesses the other. Under this method there are never any simultaneous memory conflicts, so neither side of the bridge ever waits for an access.

As an option, the software programmer has the ability to use a set of semaphore registers. These can be used to determine which side of the bridge 'owns' a particular buffer. The semaphore does not lock the memory buffer, but rather provides a mechanism for software to determine if the particular memory buffer is being used. If the semaphore is not used, or is disregarded by software, the associated buffer arbitration still operates normally.

The first seven *memory buffers* have an associated semaphore *register* called SEM_BUF_(A-G). These are classified as type SREG, meaning that they also operate as a shared resource with its own hardware arbiter. The arbitration of SMEM and SREG types are identical. An eighth buffer, 'BUF_H', does not have an semaphore register associated with it.

2.4.2 Hardware Arbitration

Accesses from both the VMEbus and PCI side of the bridge proceed normally when they are uncontested. When simultaneous accesses take place in a shared memory or shared register, a hardware arbiter holds off the access from either the VMEbus or the PCI side of the bridge. This means that only one access can take place at any given time. Each SMEM or SREG contains an identical arbiter.

If a simultaneous access occurs from both the VMEbus and PCI sides of the bridge (over a single clock period), then the hardware arbiter will grant the access to the VMEbus side of the bridge, and hold off the PCI side.

2.4.3 PCI Accesses

Each register or buffer arbiter assigns a resource at the beginning of every bus cycle. If the PCI side of the bridge wins the arbitration, then it holds the resource until it is done with its bus cycle. This means that arbitration only takes place once, at the beginning of a bus cycle. For example, if the PCI side of the bridge does a burst transfer into a shared memory, then arbitration occurs immediately before the first data transfer within the burst. The shared memory remains granted to the PCI side during the transfer. At the end of the burst transfer the shared memory is automatically relinquished[13].

---

[13] Note that behavior during burst transfers depends upon the type of memory used in SMEM. For more information see the 'Memory Requirements' section of this manual.

2.4.4 VMEbus Accesses

The VMEbus side of the bridge works somewhat differently. That side of the bridge supports posted read and write operations. During a posted read or write operation the VMEbus interface captures the data and completes any handshaking protocols with the data source (e.g. a memory buffer). At the same time it initiates handshaking with the data destination. This alleviates the need for the data source to wait until the destination is ready to accept the data.

During a VMEbus posted write operation to an SREG or SMEM region, the bridge does three things: (1) it captures the VMEbus data, (2) it asserts the DTACK* signal to begin termination of the VMEbus cycle and (3) it begins writing the data to its destination. The data is held in the posted write latch[14] until the destination arbiter is ready to accept it. When ready, the interface completes the data transfer into the SREG or SMEM region.

Once the VMEbus posted write latch has captured the data, it continues to hold it until the destination is ready to accept it. In SREG regions the data is delivered within a few clock cycles because the PCI side of the bridge is limited to single transfer cycles. However, in SMEM regions it could take some time for the data to be delivered to its destination. For example, if a 33.333 MHz PCI interface were to transfer 512 words of data during a burst transfer, then the delay before data is accepted is at least 15 uS:

$$512 \text{ WORDS/BURST} \times 1/(33.333 \text{ MHz} \times \text{WORD}) = 15 \text{ uS / BURST}$$

Once the VMEbus posted write latch has captured the data, it prevents the interface from responding to further VMEbus cycles until the data has been transferred to its destination. This prevents subsequent VMEbus cycles from overwriting the posted write data latch. The VMEbus cycle will be accepted only after the posted write data has been transferred from the latch.

Under these circumstances the waiting period could be long enough to trigger some VMEbus BERR* watchdog timers (depending upon how they are configured). However, if software uses the semaphore registers (SEM_BUF_A, etc.) to determine ownership of a buffer, then data will always be delivered within a few clock cycles.

The VMEbus posted read cycles operate in a similar, reverse manner. During read cycles from an SREG or SMEM region the VMEbus interface waits for the arbiter to grant the source to the interface. During this interval the VMEbus MASTER participating in the cycle waits until the data is ready. Once the arbiter has granted ownership of the data source to the VMEbus interface it does three things: (1) it captures the read data in the posted read latch, (2) it completes handshaking with the data source and (3) it asserts the VMEbus DTACK* signal to begin termination of the cycle. The data is held in the posted read latch until the VMEbus cycle is completed.

The posted read capability prevents the VMEbus side of the bridge from 'hogging' the SREG or SMEM area. Once arbitrated, the source data is delivered to the posted read latch in a few clock cycles. Once the data is delivered, the arbiter can service any pending accesses on the PCI side of the bridge.

---

[14] The VMEbus posted write latch is physically located in the VPWWRAP hardware section.

2.5 Reset Operation

The VME64 to PCI Bridge SoC responds to both the VMEbus [SYSRESET*] signal and the PCI [RST#] signal. Most sections of the bridge are initialized whenever the VMEbus [SYSRESET*] signal is asserted. Only the Xilinx LogiCORE PCI core is reset in response to the PCI [RST#] signal.

3.0 Hardware Reference

Most of the VME64 to PCI Bridge SoC was created and delivered in the VHDL hardware description language. VHDL source code must be synthesized before operation on a particular target device (such as an FPGA or ASIC). A variety of simulation, synthesis and CASE[15] tools can be used with the core.

Most of the components used by the core are provided with the source code. However, there are a few exceptions. RAM and I/O drivers must be synthesized with entities provided by the FPGA or ASIC vendor. That's because portable, synthesizable RAM and ROM elements are not supported by the VHDL standards. Furthermore, the SoC uses a PCI target interface that was made from a Xilinx LogiCORE PCI element. Xilinx does not provide source code for this core, but rather a 'firm core' in the form of a proprietary Xilinx '.ngo' file. This file is combined with the rest of the SoC at route time.

With the exceptions noted above, the VME64 to PCI Bridge is provided as a 'soft core'. This means that all VHDL source code and test benches are provided with the design. This enables the user to see inside of the design, thereby allowing a better understanding of it. This is useful from both a design and test standpoint. From a design standpoint the user can tweak the source code to better fit the application. From a test standpoint, it allows the user to create custom test benches that incorporate both the core and other entities on the same IC.

The soft core approach allows the VME64 to PCI Bridge to be synthesized and tested with a variety of software tools. This reduces the cost of special VHDL development software. Users should verify that their software tools conform to the IEEE standards listed in the next section of this manual.

3.1 VHDL Simulation and Synthesis Tools

It is assumed by Silicore Corporation that all simulation and synthesis tools conform to the following standards[16]:

- IEEE Standard VHDL Language Reference Manual, IEEE STD 1076-1993.
- IEEE Standard VHDL Synthesis Packages, IEEE STD 1073.3-1997.
- IEEE Standard Multivalue Logic System for VHDL Model Interoperability, IEEE STD 1164-1993.

In most cases the VHDL source code should be fully portable as long as the simulation and synthesis tools conform to these standards. However, if incompatibilities between the source code and the user's tools are found, please contact Silicore Corporation so that the problem can be resolved.

---

[15] CASE: Computer Aided Software Environment
[16] Copies of the standards can be obtained from: IEEE Service Center, 445 Hoes Lane, P.O. Box 1331, Piscataway, NJ USA 08855 (800) 678-4333 or from: www.ieee.org

It is strongly recommended that the user have a set of VHDL simulation tools before integrating the VME64 to PCI Bridge. These help in two ways: (a) they build confidence that the core synthesizes correctly and (b) they help resolve any integration problems. The simulation tools do not need to be fancy...a simple logical, non-graphical simulator is adequate.

Explicit instructions for synthesis and routing of the core on a Xilinx Spartan 2 FPGA are provided in the source disk. These take the form of batch files that exactly define the various synthesis and routing operations needed to form the final target device. The following tools are used in the source disk examples:

- Xilinx Integrated Software Environment (ISE) 4.2 (FPGA Router)
- Xilinx XST Tools (VHDL Synthesis)
- ModelSim 5.5b (VHDL Simulator)

All original VHDL source files have been edited with a MS-DOS editor. Font style: COURIER (monotype), tab spacing: 4. Almost any editor can be used, but the user may find that the style and formatting of the source code is more readable using this (or a compatible) editor.

3.2 VHDL Portability

Portability of the VHDL source code is a very high priority in the VME64 To PCI Bridge design. It is assumed that the core will be used with a variety of target devices and tools.

Several proven techniques have been used in the source code to enhance its portability. These apply to the synthesizable code, and not to the test benches. These include:

- No *variable* types are used. Variables tend to synthesize unusual logic in some VHDL compilers, and have not been used in the *synthesizable* entities. For example, all counters are designed with logic functions, and not with incremental variables. Variable types are used in the test benches, however.

- No internal three-state buses are used[17]. Some FPGA architectures do not support three-state buses well, and have been eliminated from the core (except for the I/O port interfaces, which are user defined). However, some VHDL synthesis tools will automatically create three-state buses on large multiplexors. This is perfectly acceptable if the target device supports them.

- Synchronous resets and presets are used on flip-flops. No asynchronous resets or presets are used in the design. Most FPGA and ASIC flip-flops will handle synchronous

---

[17] The Xilinx LogiCORE PCI does contain a three-state bus. Because this core contains licensing provisions which restrict its use to Xilinx parts, and because Xilinx parts do have three-state buses, this is not a problem in this application.

resets and presets very well. The asynchronous resets and presets are less portable, but are still supported by most devices. Asynchronous presets are least portable, and have been eliminated from the design.

o Asynchronous, unintended latches have been eliminated from the design. These are usually the result of incompletely specified *if-then-elsif* VHDL statements.

o Each source file contains one entity/architecture pair.. Some simulator and synthesis tools cannot handle more than one entity/architecture pair per file.

3.3 Required Resources on the Target Device

The logic resources required by the VME64 To PCI Bridge are fairly common, and are available on many FPGA and ASIC target devices. However, before synthesis the user should confirm that the following elements are available on the target device:

o At least two global, low skew clock interconnects for [PCLK] and [ECLK]. Most of the logic in the core is synchronous, and the global clocks coordinates all of the internal activity.

o Logic elements such as NAND gates, NOR gates, inverters and D-type flip-flops. Only elements defined by the IEEE STD 1164-1993 standard are used in the core.

o D-type flip-flops with known power-up conditions. The VME64 To PCI Bridge has some internal bits that must be set to pre-defined states after a power-up or configuration reset.

o Fourteen 256 x 16-bit block RAMs, and two 256 x 16-bit distributed RAMs. These are used for shared memory buffers. In order to support PCI burst transfers the RAM elements must be capable of operating within a single clock cycle. For more information see the section below regarding memory integration.

3.3.1 Clock Requirements

Three clocks are required by the VME64 to PCI Bridge. These are called [PCLK], [VCLK] and [ECLK]. [PCLK] stands for PCI CLocK, and is used by the Xilinx LogiCORE PCI IP core. [VCLK] stands for Vme CLocK, and is used by the Silicore VMEcore(tm) element. [ECLK] stands for EEPROM clock, and generates the clock for the EEPROM hardware interface. [ECLK] is formed from [PCLK]. Under certain conditions [PCLK] and [VCLK] may be tied together to form a single clock interconnection. This is done on the VMEbus to PCI Bridge Core. Table 3-1 shows the requirements for the two clocks.

33

Table 3-1. Allowable Frequency Range for [PCLK] and [VCLK].				
Clock	F_{min}	F_{max}	Duty Cycle	Controlled by
PCLK	0	33.333 MHz	40/60 – 60/40	PCI Rev 2.2
VCLK	33.333 MHz	50.000 MHz	40/60 – 60/40	VITA 1-1994 & VMEcore™

The [PCLK] clocking constraints follow those given in the PCI bus specification. However, the Xilinx LogiCORE PCI has several constraints as well. That core requires that the clock be operated at a fixed frequency[18], and not be driven by a device such as a phase lock loop (PLL). Furthermore, they do not guarantee that the core will operate correctly using industrial speed components[19].

The [VCLK] clocking constraints are governed by the Silicore VMEcore™ IP Core. That core samples an asynchronous bus, and must be provided with a clock using the constraints shown in Figure 3-1. The 33.333 MHz minimum frequency is dictated by the need to sample the asynchronous VMEbus signals fast enough to prevent aliasing problems. However, it can't go above 50.000 MHz because the period of [VCLK] must not exceed the data strobe skew on the backplane.

If [PCLK] and [VCLK] operate at (or near) 33.333 MHz, then they may be tied together. If the user believes that the manufacturing guard band on frequency is insufficient at this frequency, then it is recommended that the two cores be operated at separate frequencies.

---

[18] The PCI specification allows the PCI clock to vary as long as it remains monotonic and within the duty cycle specification.
[19] Xilinx offers countermeasures for these problems on their web site.
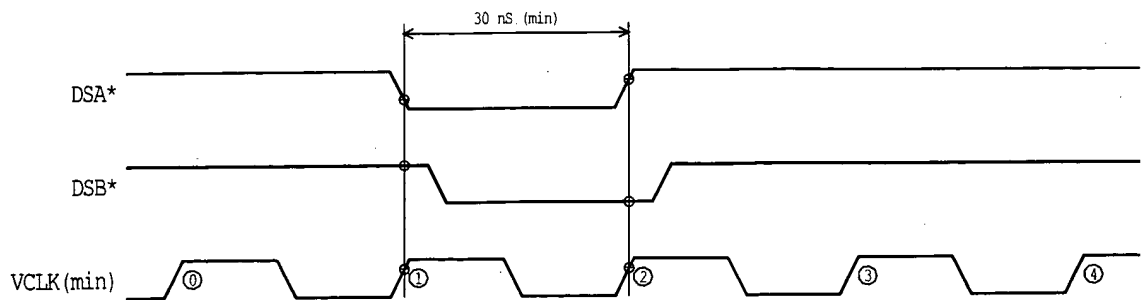
(a) AT VCLK(min) (33.333 MHz) AT LEAST ONE ASSERTED SAMPLE IS ASSURED ON BOTH DATA STROBES.
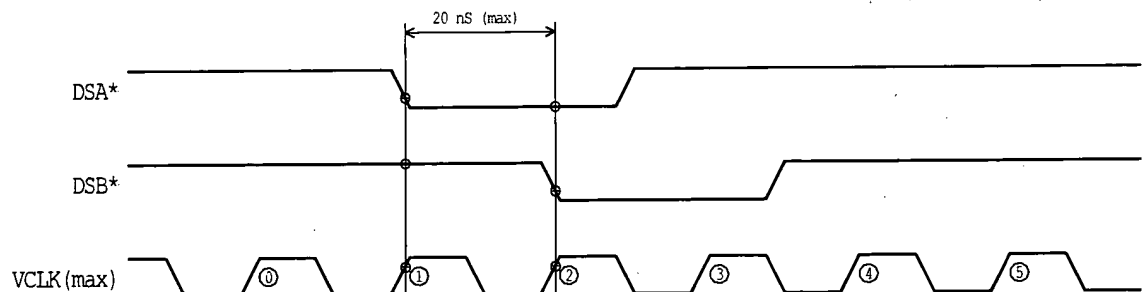
(b) AT VCLK(max) (50.000 MHz) AT LEAST ONE ASSERTED SAMPLE IS ASSURED AT MAXIMUM BUS SKEW.

JAN 24, 2002

Figure 3-1. VMEbus requirements (constraints).

3.3.2 Memory Requirements

A wide variety of RAM memory elements can be used with the design. However, some types will operate faster and more efficiently than others. In general, if the memory interface closely resembles that needed by the IP cores in the design, then everything will run fast. If the memory is significantly different, then everything will slow down.

The internal architecture of the VME64 to PCI Bridge assumes that all memories conform to something called 'FASM', or the FPGA and ASIC Subset Model[20]. That's because the Xilinx LogiCORE PCI, the Silicore VMEcore and the WISHBONE interconnection all rely on the same type of FASM synchronous RAM elements.

The FASM synchronous RAM model conforms to the generic connection and timing diagram shown in Figure 3-2. During write cycles, FASM synchronous RAM stores input data at the indicated address whenever: (a) the write enable (WE) input is asserted, and (b) there is a rising clock edge.

---

[20] The original FASM model actually encompasses many type of devices, but here the focus will be on the FASM synchronous RAM models.
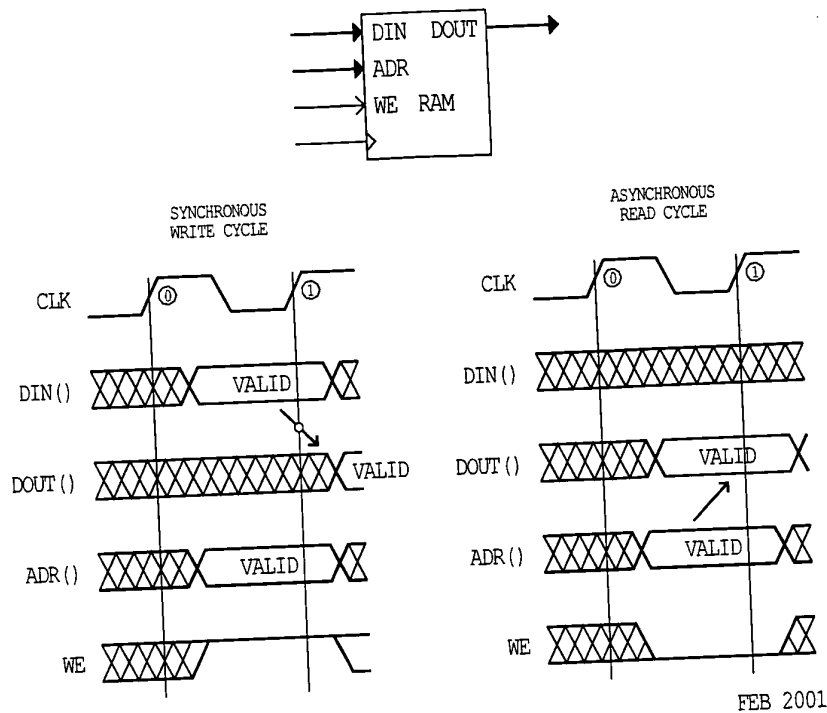
Figure 3-2. Generic FASM synchronous RAM connection and timing diagram.

During read cycles, FASM synchronous RAM works like an asynchronous ROM. Data is fetched from the address indicated by the ADR() inputs, and is presented at the data output (DOUT). The clock input is ignored. However, during write cycles, the output data is updated immediately after the rising clock edge.

The basic advantage behind the FASM synchronous RAM is its ability to work in systems that use single clock data transfer mechanisms. The limiting factor in this design is the Xilinx Logi-CORE PCI IP Core, which requires single clock data transfers during burst read and write cycles. However, this is complicated by the fact that the core does not include any provision for throt-tling the data transfers.

The Silicore VMEcore and WISHBONE interconnection systems both support single clock data transfers and data throttling.

While most FPGA and ASIC devices provide RAM that follow the FASM guidelines, many also support other types of memories too. Some of these interface smoothly to WISHBONE, while others introduce a wait-state. In all cases that we have found, all modern FPGA and most ASIC devices do support at least one style of FASM memory.

Since a Xilinx IP Core product is used in this design, and relatively large memories are needed, the SoC must support the Xilinx Block SelectRAM+ memories. Unfortunately, these do not

36

conform to the FASM guidelines, and will not work in single clock systems. They require an extra clock cycle either at the front of the cycle (to register the address) or at the back end of the cycle (to register the data). Although these can be adapted to operate in a single clock configuration (see Figure 3-3), this option was omitted in the design because the Xilinx LogiCORE PCI would not route to these memories at 33 MHz on a Xilinx Spartan 2.
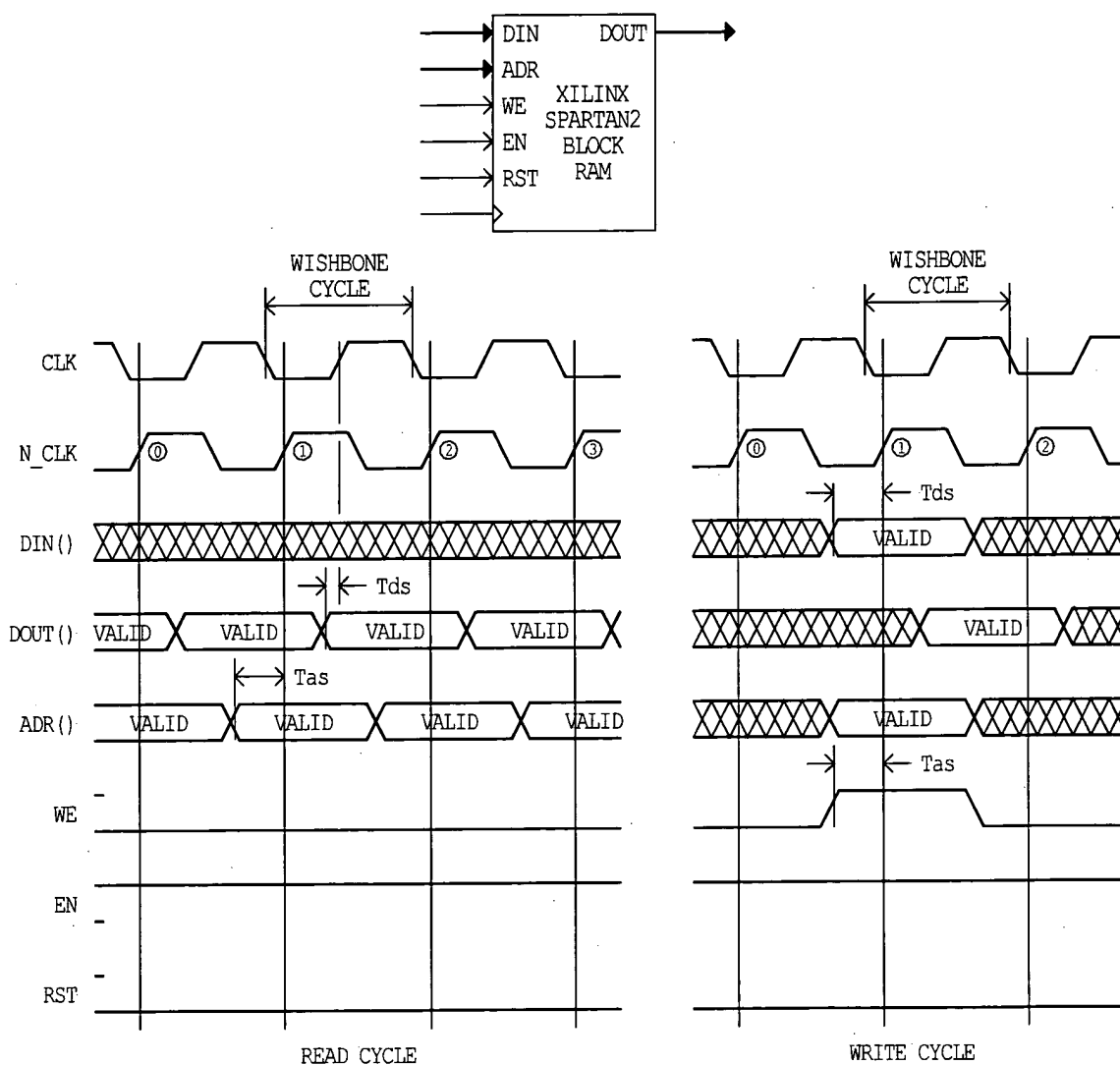
Figure 3-3. Xilinx LogiCORE PCI block RAMs when configured for single cycle operation.

The Xilinx Spartan 2 distributed RAM can support single clock bus cycles. Their main disadvantages are that they contain fewer memory bits than the Xilinx Block SelectRAM+ elements, and consume many logic LUTs (look up tables).

The internal memory buffers and circuits of the VME64 to PCI Bridge are designed to accept either single or double clock memory cycles. This is done by substituting the buffer memories for one or the other. If single clock memories are used then the burst transfers are supported on the PCI target interface. If double clock memories are used, then burst transfers are not supported. If Xilinx Block SelectRAM+ elements are used in a single clock configuration (as described above), the FPGA circuit will be more difficult to route because the clock rate of the internal circuit is effectively doubled from 33 to 66 Mhz.

Table 3-2 shows the Xilinx RAM types used for the buffer memories in the VMEbus to PCI bridge.

| Table 3-2. RAMs Used for Buffer Memories. ||
Memory	RAM Used
BUF_A	2 ea. 256 byte x 16-bit Xilinx Block SelectRAM+
BUF_B	2 ea. 256 byte x 16-bit Xilinx Block SelectRAM+
BUF_C	2 ea. 256 byte x 16-bit Xilinx Block SelectRAM+
BUF_D	2 ea. 256 byte x 16-bit Xilinx Block SelectRAM+
BUF_E	2 ea. 256 byte x 16-bit Xilinx Block SelectRAM+
BUF_F	2 ea. 256 byte x 16-bit Xilinx Block SelectRAM+
BUF_G	2 ea. 256 byte x 16-bit Xilinx Block SelectRAM+
BUF_H	2 ea. 256 byte x 16-bit Xilinx Distributed RAM

4.0 VHDL Entity Reference

The VME64 To PCI Bridge is organized as a series of VHDL entities. These are tied together into an entity called VMEPCIBR_SOC. All of the higher level (first and second tier) entities are described in this chapter in alphabetical order. For more information about the SoC hierarchy please refer to the VMEPCIBR_SOC entity description below.

4.1 CEEPROM Entity

The CEEPROM entity forms a stand-alone interface for programming the Atmel AT17 Series of FPGA configuration EEPROM. Figure 4-1 shows a block diagram and Figure 4-2 shows a functional diagram for the interface. The entity has a WISHBONE compatible data bus interfaces with characteristics shown in Table 4-1.

CEEPROM.VHD: HIGH LEVEL BLOCK DIAGRAM
17 AUG 2002
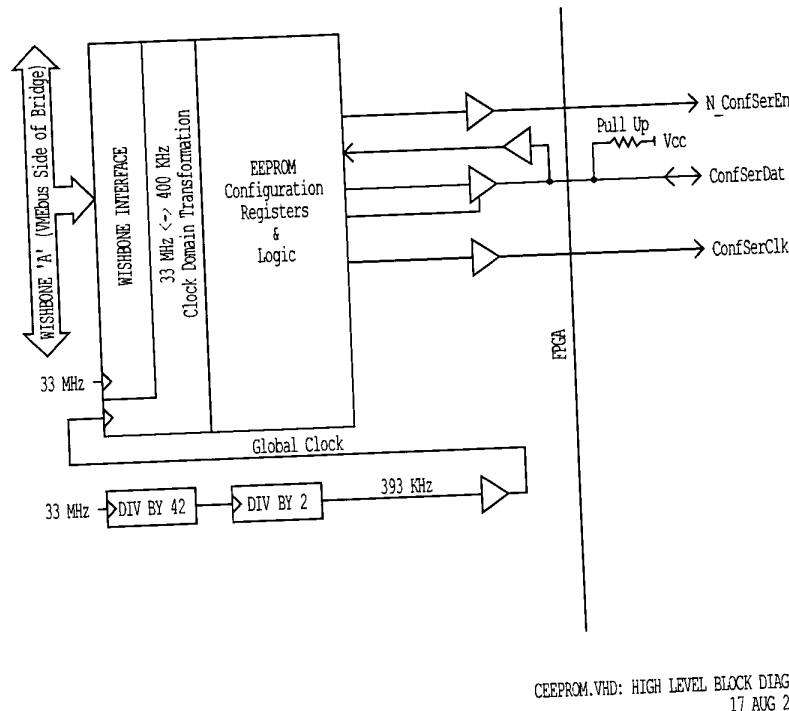
Figure 4-1. Block diagram of the configuration EEPROM Interface.

Data transactions over the data interface are synchronized to the WISHBONE clock [CLK_I]. The entity also includes a clock divider circuit to generate an EEPROM control clock called [ECLK_O]. The nominal frequency of [ECLK_O] is 393 KHz when the [CLK_I] is operated at 33 MHz. The clock divider value can be adjusted to any frequency by editing the VHDL entity. Any frequency of [CLK_I] and [ECLK_O] can be used as long as the frequency of [CLK_I] is at least four times the frequency of [ECLK_O].

EEPROM control clock [ECLK_O] is routed out of the module and then back in again via clock signal [ECLK_I]. This allows the [ECLK_O] signal to be routed to the highest level system entity so that it can drive a global clock net. This allows a global clock net buffer to reside at the highest system level for convenience.

Table 4-1. WISHBONE DATASHEET for the CEEPROM Entity	
General Description	SLAVE interface for the Atmel AT17 series of FPGA configuration EEPROM.
WISHBONE Revision Level	B.2
Supported WISHBONE Cycles	SLAVE: READ/WRITE
Data port, size	32-bit
Data port, granularity	16-bit
Data port, maximum operand size	32-bit
Data transfer ordering	BIG ENDIAN or LITTLE ENDIAN
Data transfer sequencing	None
Signal Description	All WISHBONE signal names are identical to those defined in the specification.
Terminating Signals	The WISHBONE interface on both ports support only the [ACK_O] terminating signal.
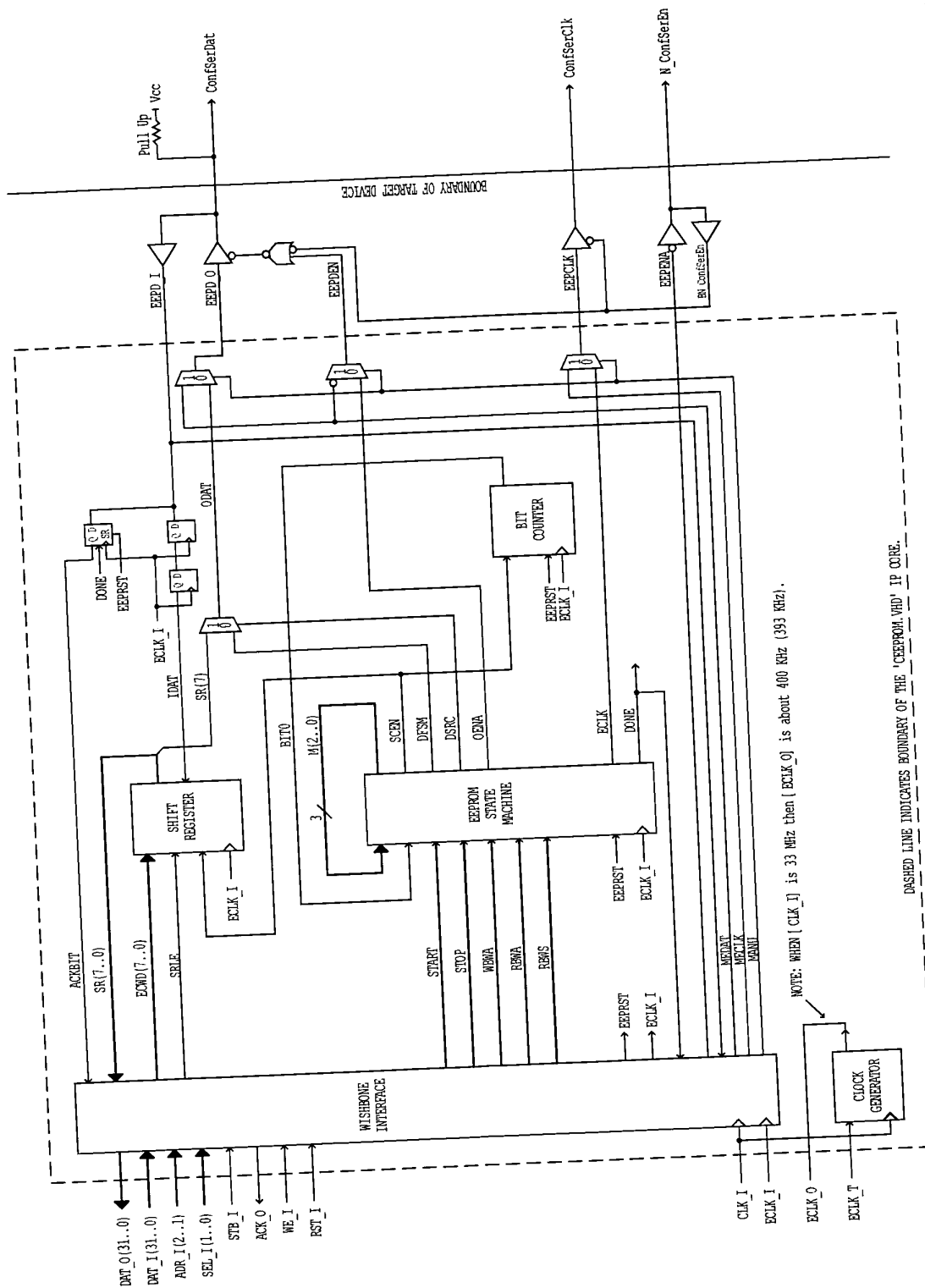
Figure 4-2. Functional diagram of CEEPROM.VHD.

42

The [EEPENA] signal is asserted whenever the 'ConfSerEn' bit (D07) is set in the CONFIG_PROM_CMD register. This indicates that the EEPROM is enabled for configuration. It also indicates that the three-state clock and data pin drivers on the target device are enabled. External hardware (other than the EEPROM itself) must refrain from driving these signals when [EEPENA] is asserted.

The WISHBONE interface includes all registers that are accessible through the interface. They operate at the nominal WISHBONE clock speed of 33 MHz. The WISHBONE interface also includes a clock domain transformation circuit as shown in Figure 4-3(a). Figure 4-3(b) describes the relationship between the two clock domains. The clock transformation circuit allows two clock domains to communicate seamlessly. Figure 4-4 shows a timing diagram for the circuit.

The EEPROM state machine controls all low frequency activity for the EEPROM interface, with its state diagram shown in Figure 4-5. The state machine forms an instruction set with six instructions. Five of these instructions correspond to the bits in the CONFIG_PROM_CMD register described elsewhere in this manual. A sixth instruction (WAIT) is a type of NOP (no operation). When an instruction is requested (e.g. SendStartCondition), the state machine generates four 'RISC type' instructions. These, in combination with some control bits, handshakes with the EEPROM. A loop instruction is formed with a the BIT_COUNTER process. Figures 4-6 and 4-7 shows the high-level timing for the EEPROM.

PROCESS: EEPC_REGISTER   PROCESS: SYNC_COMMAND
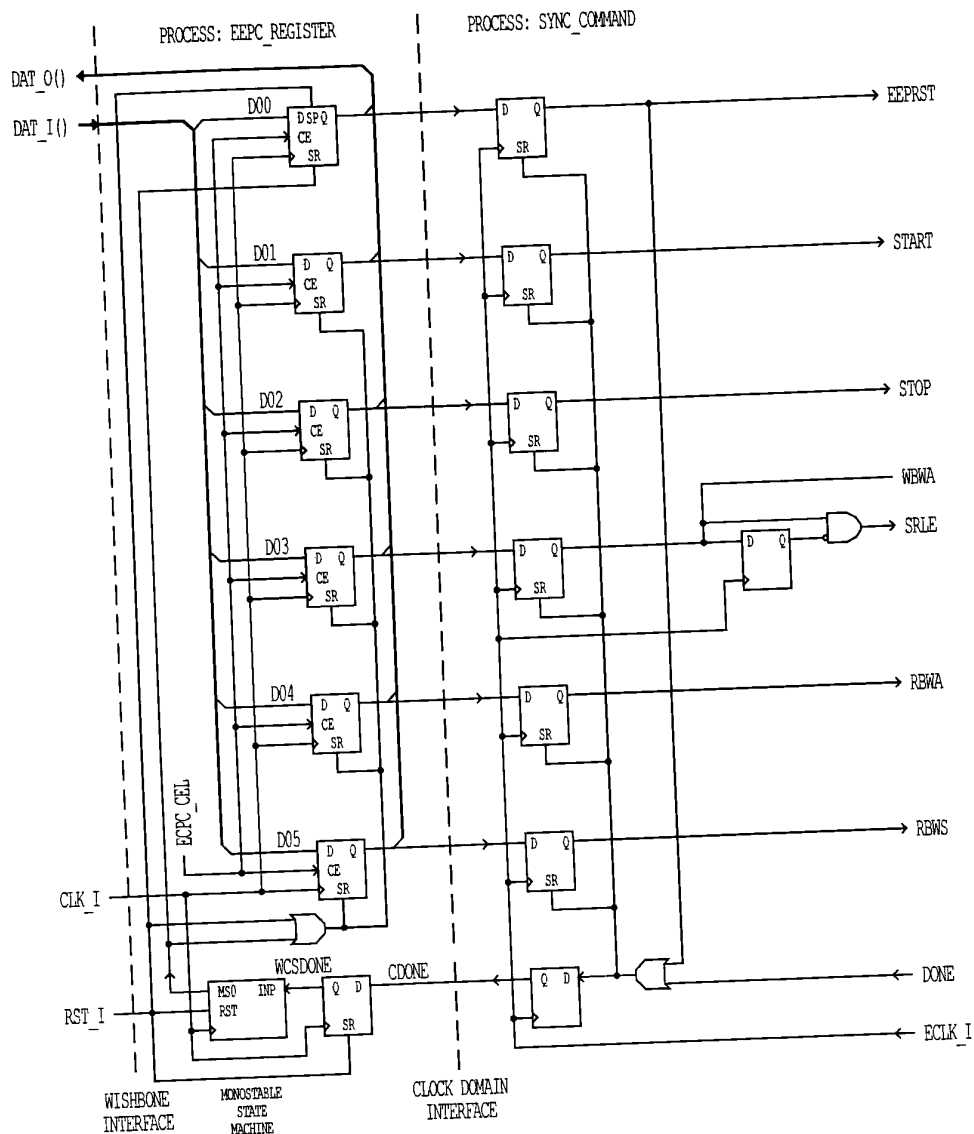
MONOSTABLE STATE MACHINE

STATES: MS1, MS0
INPUTS: WCSDONE

EQUATIONS:

MS0 = /MS1 * /MS0 * WCSDONE;

MS1 = /MS1 * MS0 * WCSDONE
    + MS1 * /MS0 * WCSDONE;
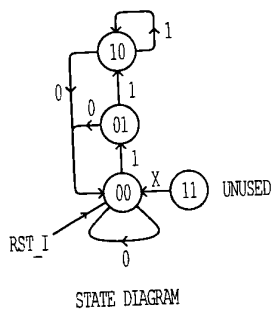
STATE DIAGRAM

CEEPROM.VHD: HANDSHAKING CIRCUIT
20 AUG 2002

Figure 4-3(a). Clock domain transformation circuit.

44

NOTE: FOR PROPER OPERATION OF EACH BIT,THE FREQUENCY OF 'CLK_I'
MUST BE AT LEAST FOUR TIMES THE FREQUENCY OF 'ECLK'.

BECAUSE:

```
   1 CLK_I FLIP-FLOP SET-UP
   1 CLK_I SYNCHRONIZATION
 + 1 CLK_I MONOSTABLE STATE MACH.
 + 1 CLK_I DATA FLIP-FLOP
 + 2 CLK_I WISHBONE RMW CYCLE
   ----------------------------------------------
 = 6 CLK_I OVERHEAD        PER 1 ECLK
 + 1 ECLK  FLIP-FLOP SET-UP PER 1 ECLK
```

FURTHERMORE, IT IS RECOMMENDED THAT THE FREQUENCY OF [CLK_I] BE SUFFICIENTLY
HIGHER THAN THE FREQUENCY OF [ECLK] TO ALLOW FOR A
HOST PROCESSOR TO READ AND WRITE TO THE BITS AND
RESPOND ACCORDINGLY.

THE MAXIMUM RESPONSE TIME OF THE HOST PROCESSOR CAN BE FOUND THUSLY:

$$Tav = \frac{1}{ECLK} - \frac{6}{CLK\_I} - 1\ ECLK\_MAX\_FF\_SET\text{-}UP$$

FOR EXAMPLE:

```
  F, CLK_I = 33.000 MHz
  F, ECLK  =  0.400 MHz
```

$$Tav = \frac{1}{0.4\ MHz} - \frac{5}{33\ MHz} - \frac{1}{33\ MHz} = 2.32\ uS$$

FOR MORE INFORMATION, SEE THE TIMING DIAGRAM.

<div style="text-align:right">CEEPROM.VHD: HANDSHAKING CIRCUIT
20 AUG 2002</div>

Figure 4-3(a). Clock domain transformation circuit (con't).
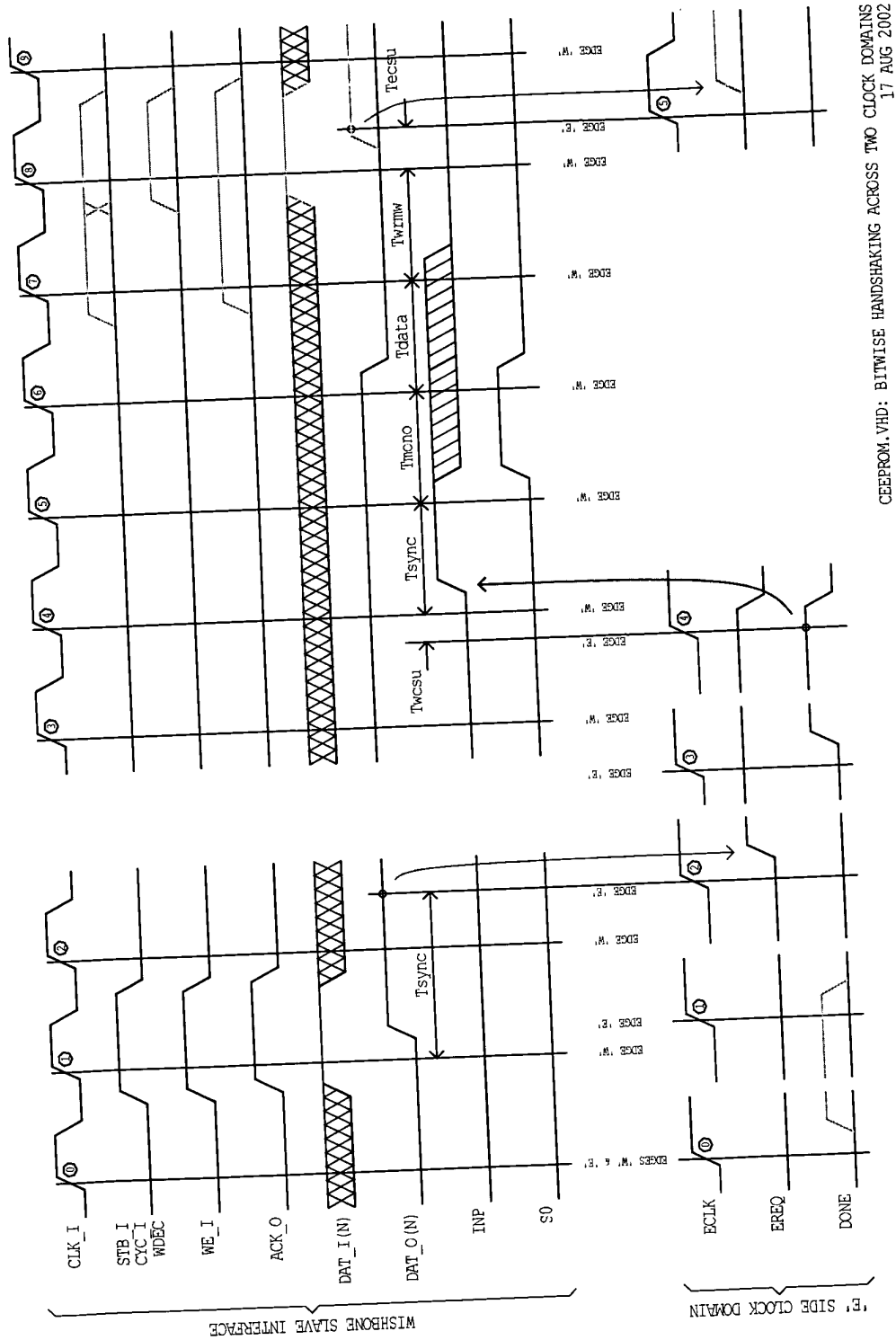
Figure 4-4. Timing for the clock domain transformation circuit.

46
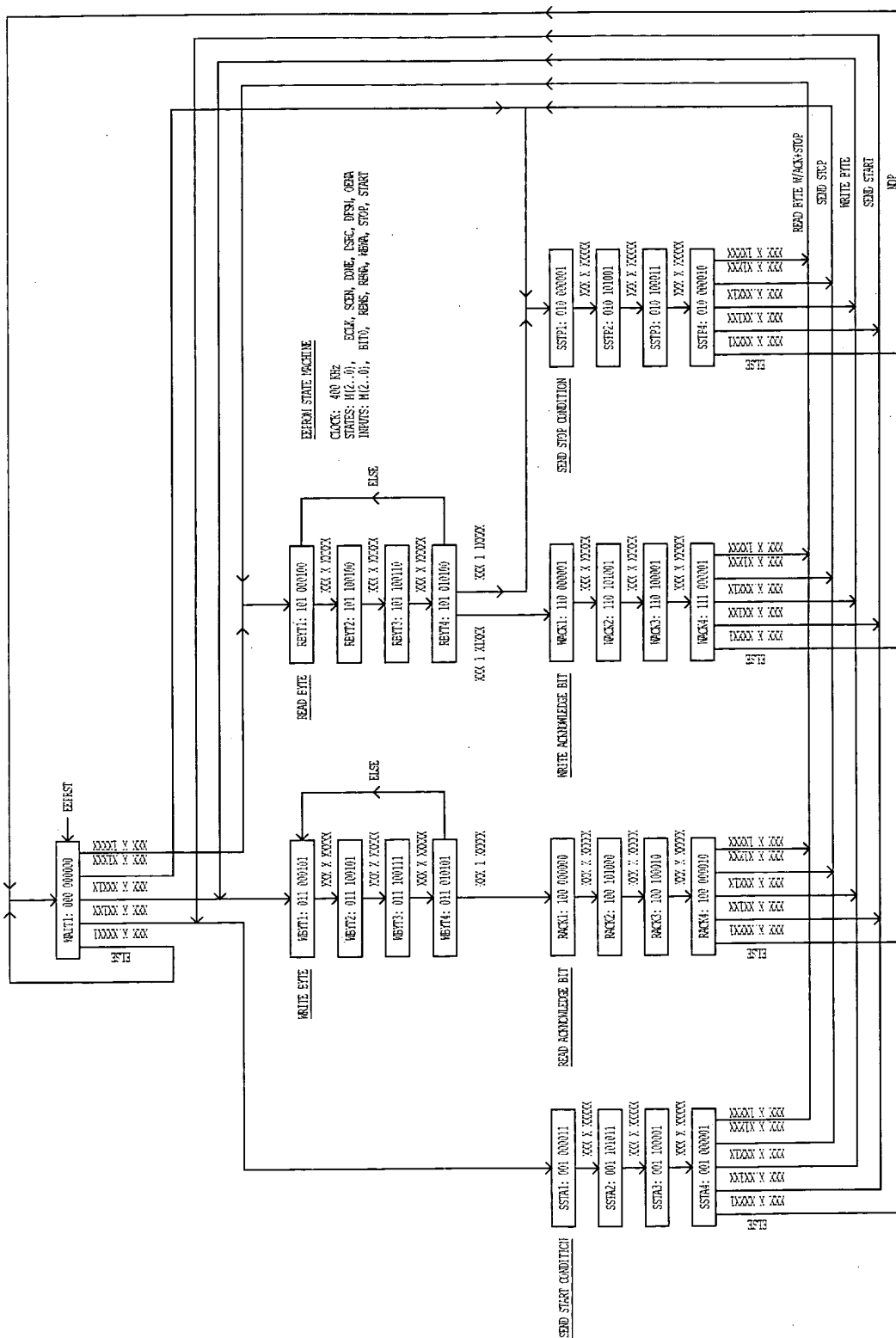
Figure 4-5. State diagram for EEPROM State Machine.

Figure 4-6. EEPROM START and STOP condition timing.

48

Figure 4-7. EEPROM read and write byte timing.

49

4.2 MISCREG Entity

The MISCREG entity contains all logic for controlling the lower seven registers in the VMEP-CIBR address map. It contains two WISHBONE interfaces named 'A' and 'B'. In the VMEP-CIBR SoC the 'A' side is connected to the WISHBONE interconnection served by VMEbus, and the 'B' side is connected to the interconnection served by the PCI interface. The entity also contains all the arbitration logic necessary for handling accesses from the dual 'A' and 'B' ports.

Figure 4-8 shows a block diagram of the MISCREG entity, and Table 4-2 shows the WISHBONE DATASHEET for the interfaces. The seven registers handled by the entity include:

- DMC_HW_CONTROL
- CONFIG_PROM_CMD (see the CEEPROM entity)
- CONFIG_WRITE_DATA (see the CEEPROM entity)
- CONFIG_READ_DATA (see the CEEPROM entity)
- DMC_CMD
- DMC_FAULT
- DMC_STATUS
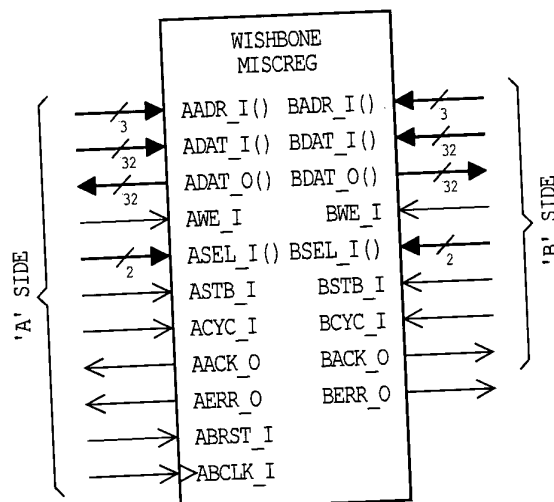
BLOCK DIAGRAM

15 APR, 2002

Figure 4-8. MISCREG block diagram.

The MISCREG arbiter and arbiter timing are shown in Figures 4-9 and 4-10 respectively.

Table 4-2. WISHBONE DATASHEET for the MISCREG Entity	
General Description	Logic for the lower seven registers in the system. Contains dual WISHBONE ports named 'A' and 'B'. This datasheet applies to both interfaces.
WISHBONE Revision Level	B.2
Supported WISHBONE Cycles	SLAVE: READ/WRITE
Data port, size	32-bit
Data port, granularity	16-bit
Data port, maximum operand size	32-bit
Data transfer ordering	BIG ENDIAN or LITTLE ENDIAN
Data transfer sequencing	None
Signal Description	All WISHBONE signal names are identical to those defined in the specification, except that they have an 'A' or 'B' at the front. The 'A' and 'B' refer to PORT A and PORTB respectively.
Terminating Signals	The WISHBONE interface on both ports support [ACK_O] and [ERR_O] terminating signals. [ERR_O] is generated when accesses to the unused/reserved registers are attempted.

'A' SIDE GRANT

UNUSED 11

01

00 X1

ABRST_I

00 00 10 01

XX

00 10

10

INPUTS: BCYC_I, ACYC_I
STATES: BGNT, AGNT
CLOCK: BCLK

'B' SIDE GRANT

1X

'A' SIDE PRIORITY
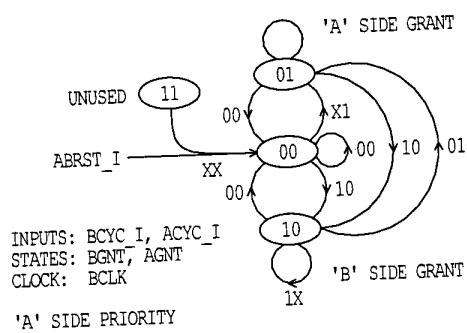
ARBITER EQUATIONS

AGNT   = /ABRST_I * /BGNT   * ACYC_I
       + /ABRST_I * /BGNT * /BCYC_I * ACYC_I
       + /ABRST_I * /AGNT * /BCYC_I * ACYC_I;

BGNT   = /ABRST_I * BGNT * /AGNT * BCYC_I
       + /ABRST_I * /BGNT   * BCYC_I * /ACYC_I;

ARBITER STATE MACHINE

15 APR, 2002

Figure 4-9. MISCREG arbiter.

52

MISCREG ARBITRATION TIMING
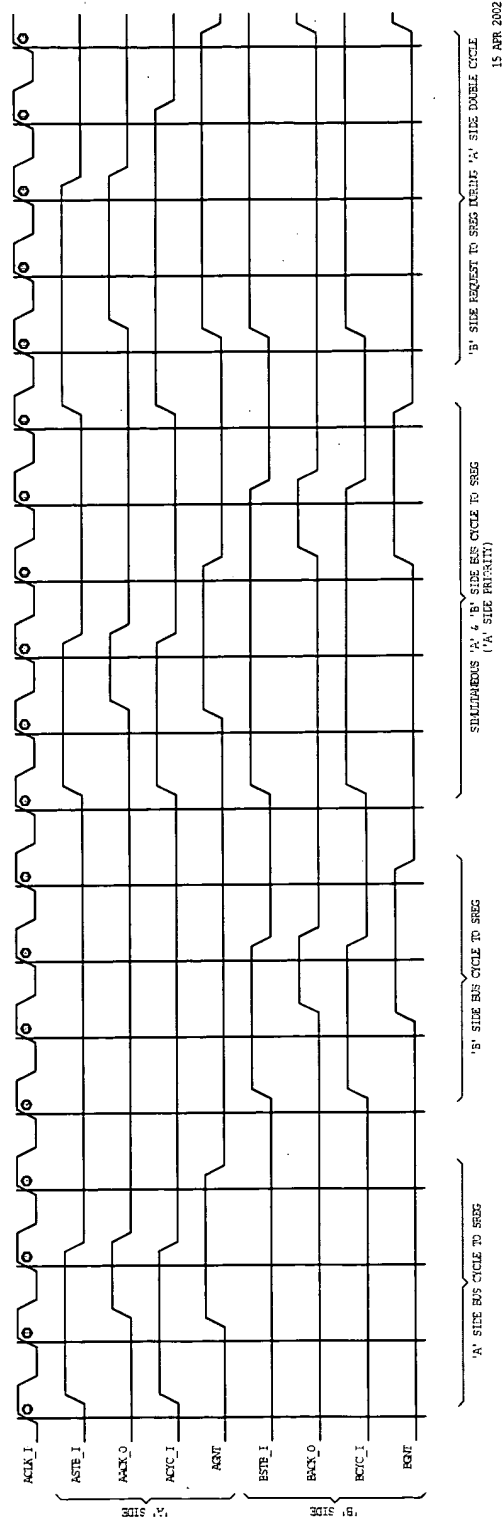
Figure 4-10. MISCREG arbiter timing.

4.3 PCIWRAP Entity

The PCIWRAP entity is a wrapper between the Xilinx LogiCORE(tm) PCI 32-bit 33 MHz IP Core and the WISHBONE SoC interconnection. As shown in the system block diagram of Figure 4-11, the wrapper establishes a PCI interface to the SoC. The wrapper is synthesized from the 'PCIWRAPc.VHD' file.

20 SEP 2002
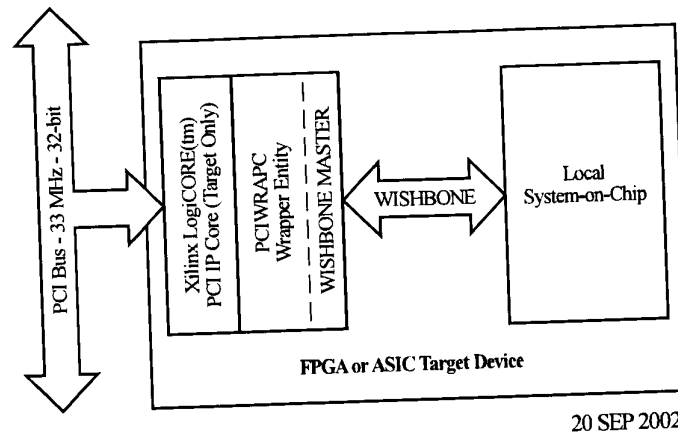
Figure 4-11. System block diagram showing the PCIWRAPc wrapper.

Figure 4-12 shows a functional diagram of the PCIWRAPc entity. Each box in the diagram corresponds to a VHDL process (as described below) of the same name.

The left side of the block diagram shows the signals connected to the PCI core. These are synthesized, routed and connected to the PCI core. The PCI core is delivered as a 'black box', so the signals must conform to the Xilinx specifications. The reader is directed to the Xilinx LogiCORE™ PCI Design Guide (Ver 3.0 – March 16, 2002 or later) for a complete description of the PCI core and signal names.

The PCI target interface responds to the following PCI commands (cycles):

- Configuration Read (CBE[3:0] = 1010)
- Configuration Write (CBE[3:0] = 1011)
- Memory Read (CBE[3:0] = 0110)
- Memory Write (CBE[3:0] = 0111)
- Memory Read Multiple (CBE[3:0] = 1100)
- Memory Read Line (CBE[3:0] = 1110)

The PCI wrapper supports BYTE granularity. However, the WISHBONE interface can be easily configured for WORD and DWORD granularity as well.

54

All PCI accesses to 'unused' address areas are terminated with a PCI TARGET ABORT.

The wrapper circuitry connected to Xilinx LogiCore PCI interface controls how the target interface responds to the PCI commands. During the initial phase of a PCI burst cycle, a binary counter (located inside the wrapper circuitry) latches the starting PCI address. The counter increments during subsequent phases within the burst cycle, thereby generating the next address. The counter is incremented after every cycle that accesses the high byte of a 32-bit DWORD transfer. The high byte is indicated when [S_CBE(3)] is low. That means that the address counter is incremented after every 32-bit transfer or after a high order 16-bit transfer.

PCI single and burst transactions are supported by the wrapper. However, burst transactions[21] may not necessarily be supported by all memory locations or registers. If a burst transaction is attempted in a memory region that does not support burst transfers, then the memory should respond with the WISHBONE [ERR_I] signal, which causes the wrapper to respond with a TARGET ABORT termination[22].

The wrapper is implemented as a PCI target, meaning that it cannot initiate PCI transactions.

The right side of the block diagram shows the signals connected to the WISHBONE MASTER interface. The interface supports 8, 16 and 32-bit data transfers and a 32-bit address bus. Table 4-3 is the WISHBONE DATASHEET that specifies the interface. The reader is directed to the WISHBONE specification, revision B.2 for a complete description of the WISHBONE MASTER and related signal names.

---

[21] Burst transactions are bus cycles with more than one data transfer phase.
[22] Burst transactions in excess of one data transfer are allowed as long as the memory structure supports it. This is because the core can be implemented on a number of target devices, memory types and speeds. The Xilinx LogiCore PCI requires that memories must support single clock data transfers. If this type of memory is implemented on the target device, then the burst operation is supported. If this type of memory is not implemented on the target device, then burst transactions are not supported. For more information please refer the Hardware Reference section of this manual.
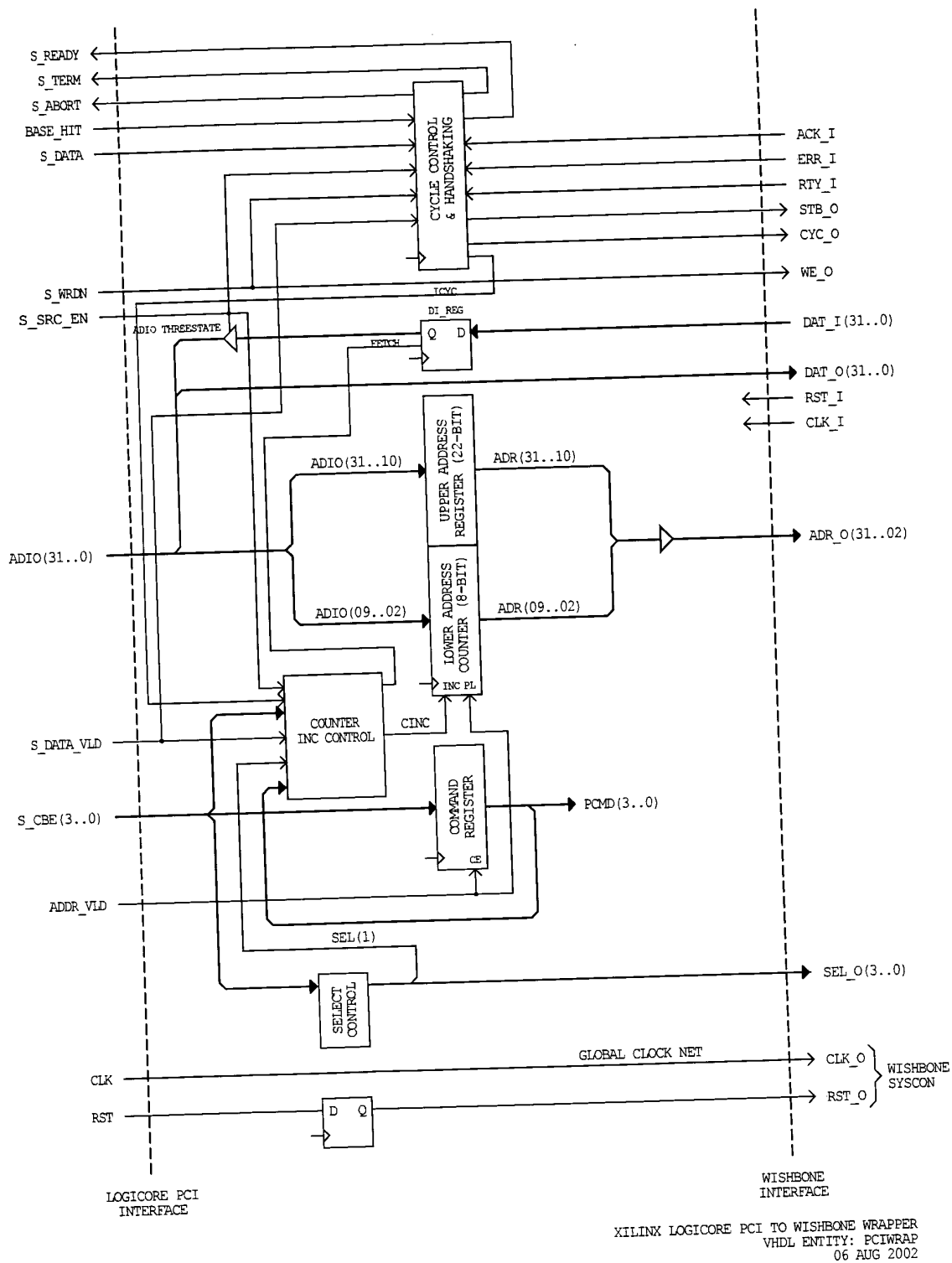
Figure 4-12. Functional diagram of the PCIWRAPC entity.

Table 4-3. WISHBONE DATASHEET for the PCIWRAP Interface.	
General Description	Wrapper between a 32-bit Xilinx Logi-CORE(tm) PCI interface and WISHBONE SoC interface.
WISHBONE Revision Level	B.2
Supported WISHBONE Cycles	MASTER: READ/WRITE MASTER: BLOCK READ/WRITE
Data port, size	32-bit
Data port, granularity	8-bit
Data port, maximum operand size	32-bit
Data transfer ordering	LITTLE ENDIAN
Data transfer sequencing	Sequential (burst) data transfers must be made from lower to higher addresses. The internal address counter rolls over whenever a most significant byte is transferred.
Signal Description	All WISHBONE signal names are identical to those defined in the specification. Refer to the signal descriptions for more details.
Terminating Signals	The WISHBONE interface supports all three termination signals: [ACK_I], [RTY_I] and [ERR_I]. See the text for more information about the operation of these signals.

4.3.1 PCIWRAP Timing Conversion

PCIWRAP is a VHDL 'wrapper', which means that it is a chunk of code that converts the PCI core signals to the WISHBONE MASTER signals. Figures 4-13 and 4-14 show the relationship between the PCI bus signals and the Xilinx PCI core back end (user application) signals. Diagrams for burst read and write cycles are shown. Figure 4-15 and 4-16 show the relationship between the PCI core back end (user application) signals and the WISHBONE MASTER interface signals. Diagrams for read and write cycles are shown.

PCI Bus to LogiCORE(tm) (User Application) Signals - PCI MEMORY BURST READ TO WISHBONE BLOCK READ - PARTICIPATING TARGET W/NORMAL TERMINATION
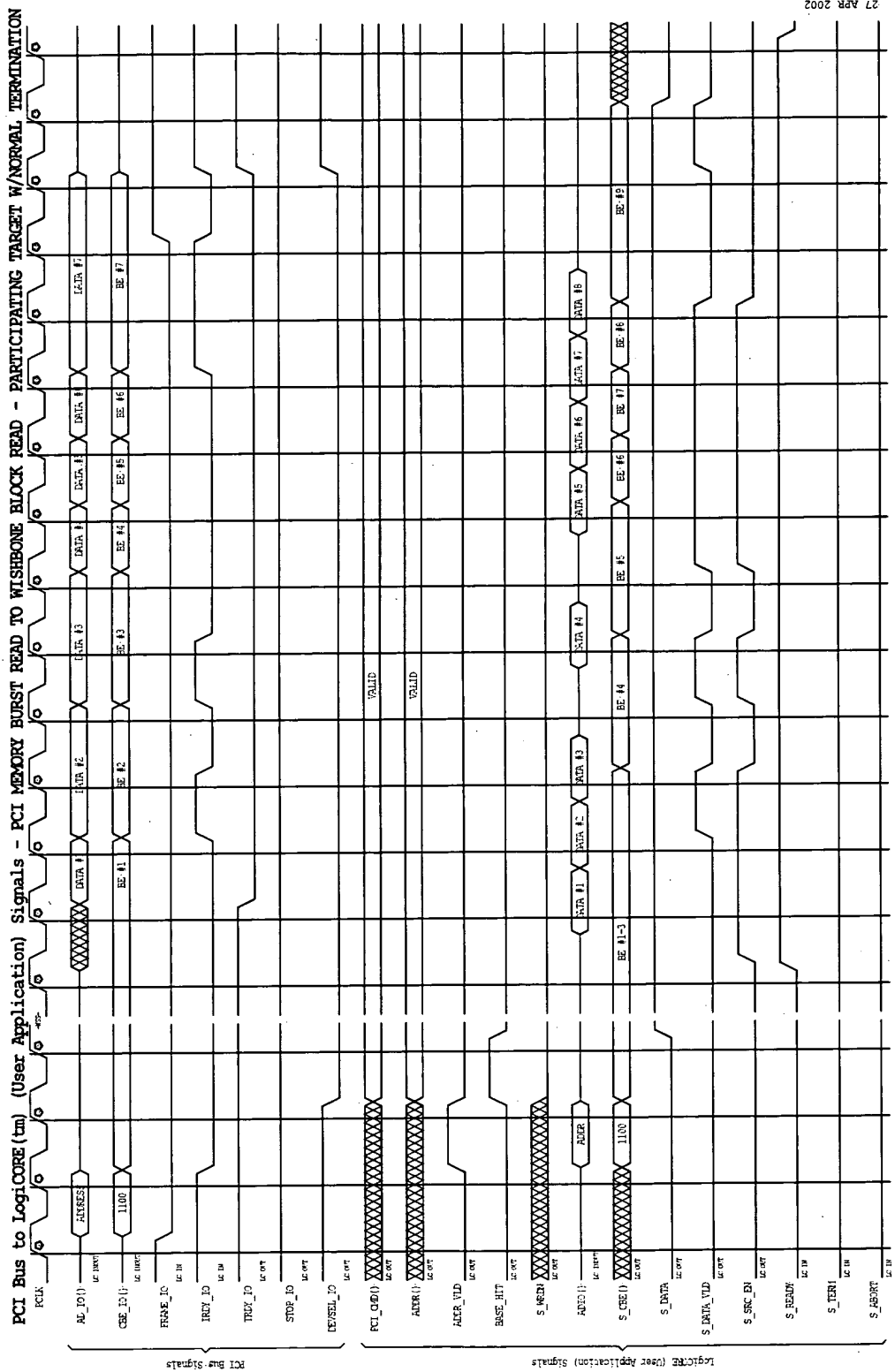
Figure 4-13. READ timing between PCI and Xilinx PCI core back end signals.
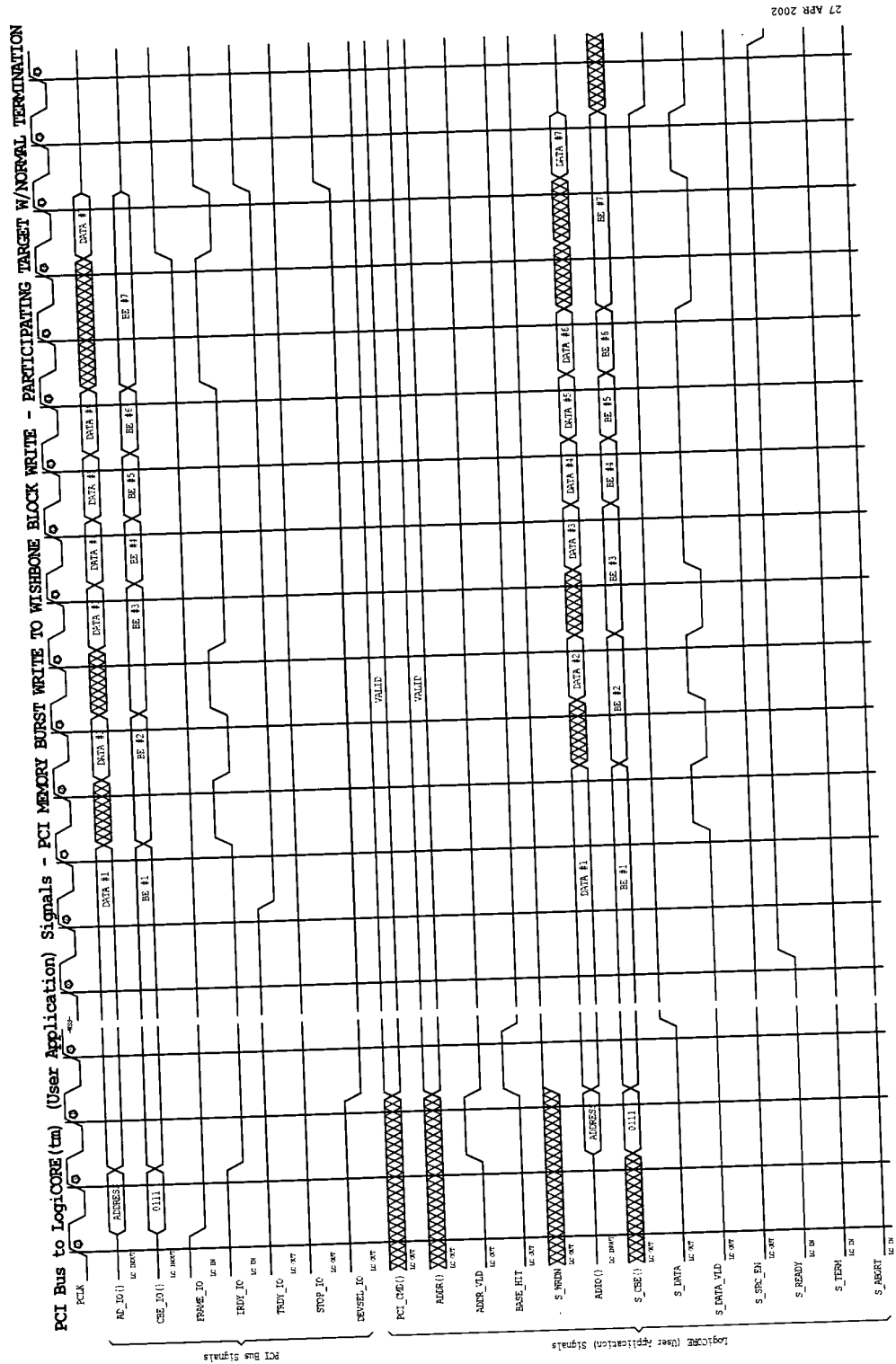
Figure 4-14. WRITE timing between PCI and Xilinx PCI core back end signals.
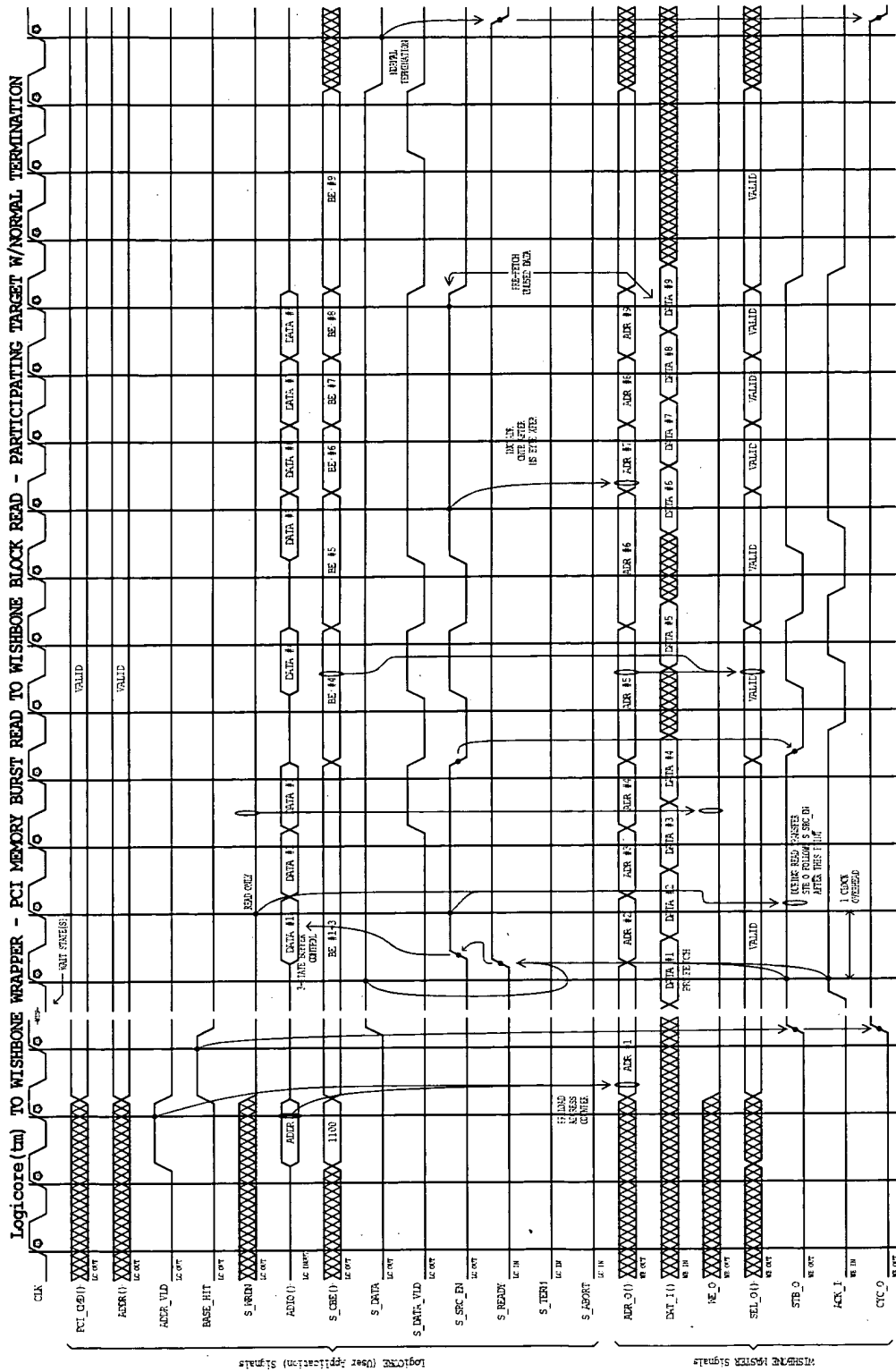
Figure 4-15. PCI memory burst read cycle to WISHBONE block read.

Figure 4-16. PCI memory burst write cycle to WISHBONE block write.
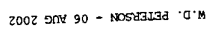
62

4.3.2 Operation of the WISHBONE Termination Signals

The PCIWRAP entity supports all three WISHBONE termination signals: [ACK_I], [RTY_I] and [ERR_I]. These are translated to control signals used by the Xilinx LogiCORE(tm) PCI core. The wrapper responds differently to these signals, depending upon when they are asserted during the PCI bus cycle.

Also note that the WISHBONE specification requires that only one termination signal be applied at any given time. Stated another way, [ACK_I] and [ERR_I] (or other combinations) cannot be asserted at the same time. Assertion of two or more acknowledge signals at any given time could result in an undefined operation.

4.3.3 PCI Normal Termination Using [ACK_I]

Figure 4-17 shows the timing diagram for a normal termination to a participating PCI TARGET cycle. There, the WISHBONE MASTER cycle starts in response to the assertion of BASE_HIT() by the Xilinx LogiCORE PCI (not shown). The PCIWRAPC wrapper then waits for the participating WISHBONE SLAVE to respond by asserting its [ACK_I] signal.
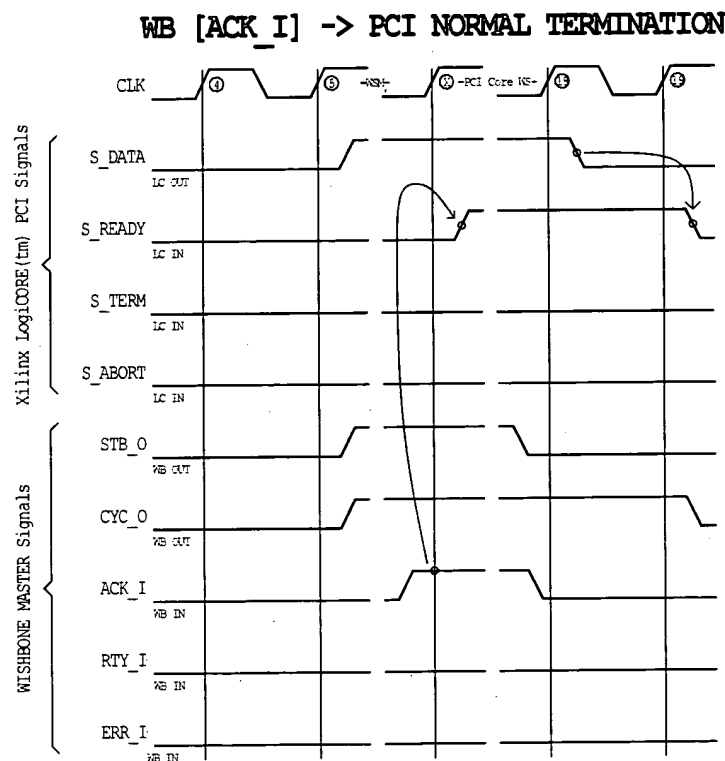
Figure 4-17. PCI normal termination.

In theory, the WISHBONE SLAVE can insert any number of wait states before responding to the cycle. However, the Xilinx documentation states that it must respond within 16 clock cycles[23]. This means that any arbitration on the WISHBONE SLAVE must be completed during this time.

It is also recommended that any WISHBONE SLAVEs be designed so that all arbitration is performed in response to the assertion of [CYC_O] by the MASTER (rather than [STB_O]). Once arbitration has been completed, the SLAVE should be made available to respond immediately to multiple assertions of [STB_O] by the MASTER. Stated another way, the SLAVE should only arbitrate at the beginning of a WISHBONE bus cycle. This allows immediate access to the SLAVE's resources throughout PCI burst transfers (if supported).

Once the WISHBONE MASTER interface has received the [ACK_I] signal, it asserts the [S_READY] signal to the PCI core. This signal remains asserted for the entire duration of the bus cycle, regardless of the state of the [ACK_I] signal. However, if the [RTY_I] or [ERR_I] signals are asserted, then the wrapper will generate either a PCI TARGET DISCONNECT W/O DATA or a TARGET ABORT.

During PCI burst transfers the [S_READY] signal remains asserted until the end of the PCI burst cycle. After the first cycle the wrapper does not respond to the [ACK_I] signal. This is because the PCI core requires that the WISHBONE SLAVE must supply or latch data during every clock cycle. Stated another way, the Xilinx PCI core does not support any handshaking mechanisms to throttle the speed of the transfer during this time.

4.3.4 PCI Target Retry Termination Using [RTY_I]

Figure 4-18 shows the timing diagram for a TARGET RETRY termination to a participating PCI TARGET cycle. There, the WISHBONE MASTER cycle starts in response to the assertion of BASE_HIT() (not shown) by the PCI core. The PCIWRAPC wrapper then waits for the participating WISHBONE SLAVE to respond. Normally, the WISHBONE SLAVE responds by asserting the [ACK_I] signal. However, if the SLAVE responds with [RTY_I], then the PCIWRAPC wrapper asserts [S_TERM] to the PCI core, which results in one of two behaviors:

- If the cycle is a burst cycle and the [ACK_I] had not been previously asserted, then the assertion of the [RTY_I] signal results in termination of the PCI bus cycle with a TARGET RETRY. The same reaction happens if the cycle is a non-burst cycle.

- If the [ACK_I] had been previously asserted during the current bus cycle, then the assertion of the [RTY_I] signal results in the termination of the PCI bus cycle with a TARGET DISCONNECT W/O DATA.

---

[23] The allowed delay is probably less than 16 clock cycles. The 16 clock cycle delay is evidently imposed by the PCI timer. However, there is probably some overhead added by the Xilinx LOGIcore(tm) PCI core.

WB [RTY_I] -> PCI TARGET RETRY
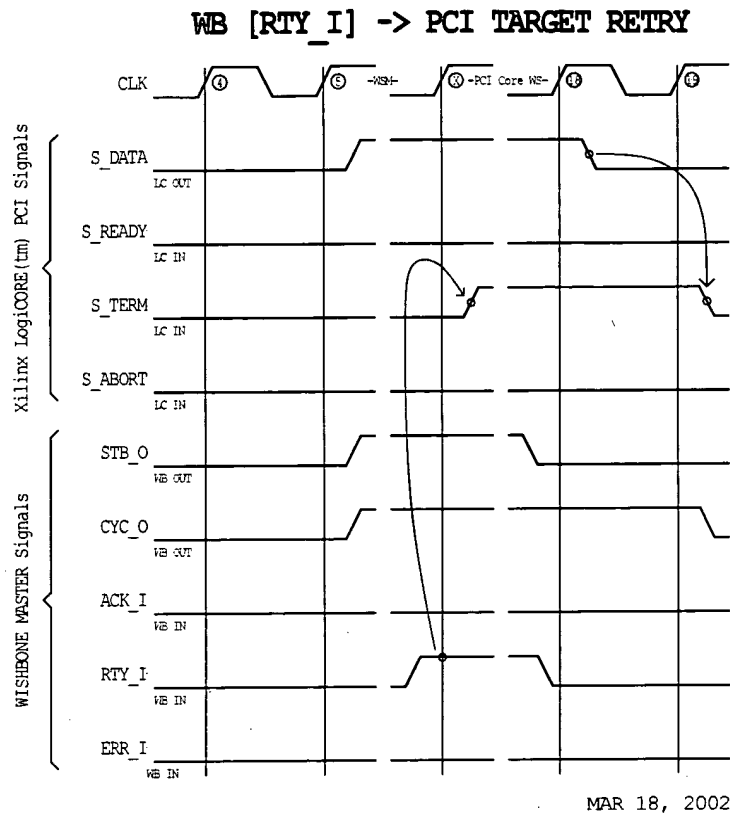
Figure 4-18. PCI target retry termination.

4.3.5 PCI Target Abort Termination Using [ERR_I]

Figure 4-19 shows the timing diagram for a TARGET ABORT termination. There, the WISHBONE MASTER cycle starts in response to the assertion of BASE_HIT() (not shown) by the PCI core. The PCIWRAPC wrapper then waits for the participating WISHBONE SLAVE to respond. Normally, the WISHBONE SLAVE responds by asserting the [ACK_I] signal. However, if the WISHBONE bus cycle is terminated by asserting the [ERR_I] signal, then the wrapper asserts [S_ABORT] to the PCI core. This results in a TARGET ABORT termination. The TARGET ABORT can be generated at any time during a single or burst transfer.
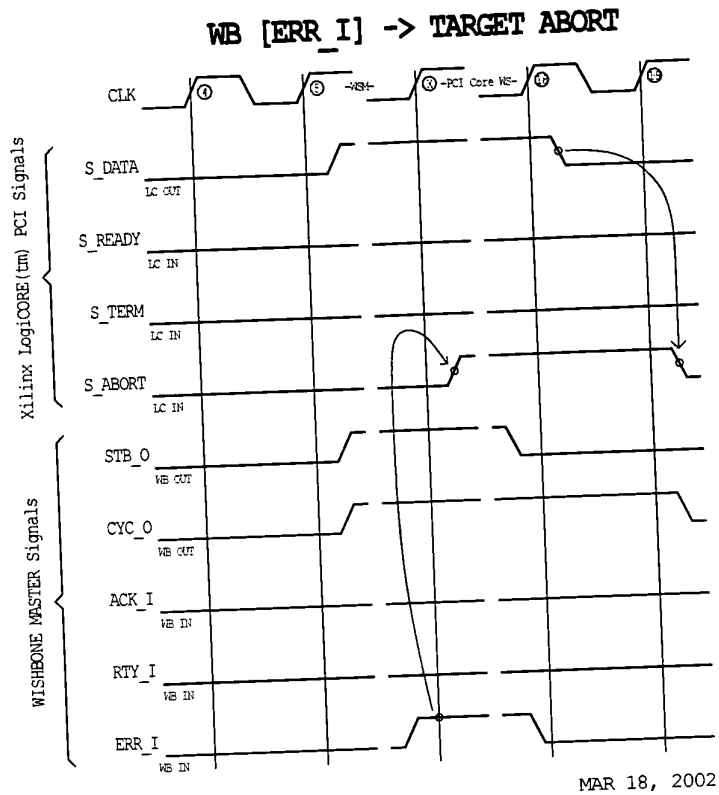
Figure 4-19. PCI target abort termination.

4.3.6 VHDL Entity Reference

The PCIWRAP entity is synthesized from a top level file called 'PCIWRAPC.VHD'. Each module of VHDL code is named with a seven character 'handle' that is related to its entity name. An additional (final) character is added to indicate it's use. A 'C' character indicates that it's a VHDL circuit file, a 'T' character indicates that it's a test bench file, and a 'V' character indicates that it's a test vector file (with a '.txt' file extension). The 'PCIWRAP' entity/architecture has the following names associated with it:

Entity (circuit) name: PCIWRAPC
Architecture name: PCIWRAPC1

Entity/architecture filename: PCIWRAPC.VHD
Test bench filename: PCIWRAPT.VHD
Text vector filename: PCIWRAPV.TXT

66

4.3.7 Process: ADIO_THREESTATE

The ADIO_THREESTATE process generates a three-state output buffer on the 32-bit ADIO bus.

4.3.8 Process: COMMAND_REGISTER

The COMMAND_REGISTER process creates the command register. The command register stores the state of the PCI command that is indicated on 'S_CBE()' during the initial (address) phase of every bus cycle.

4.3.9 Process: COUNTER_INC_CONTROL

The COUNTER_INC_CONTROL generates the counter increment signal 'CINC'. The counter is incremented at different times, depending on if it's a read or a write cycle. The type of cycle is determined from the 'PCMD(3..0)' register, which stores the state of the PCI command (on [C/BE[3:0]#). The counter is incremented only during Memory Read Multiple ([C/BE[3:0]# = B"1100") and Memory Write ([C/BE[3:0]# = B"0111") cycles.

The counter is incremented after every cycle that accesses the high byte of a 32-bit DWORD transfer. The high byte is indicated when [S_CBE(3)] is low. That means that the address counter is incremented after every 32-bit transfer, after a high order 16-bit transfer and a high order 8-bit transfer.

4.3.10 Process: CYCLE_CONTROL

The CYCLE_CONTROL process generates the WISHBONE [STB_O] and [CYC_O] signals. It also generates the [S_READY], [S_TERM] and [S_ABORT] signals to the Xilinx Logi-CORE(tm) PCI interface. The process uses the state machines and equations shown in Figure 4-20.

CYCLE CONTROL STATE MACHINE:

STATES: CYC, FSTB
INPUTS: PWC, PRC, S_DATA, BASE_HIT

EQUATIONS:

PRC = /S_WRDN * S_SRC_EN;
PWC = S_WRDN * S_DATA_VLD;

FSTB := /RST_I * /CYC * /FSTB   * BASE_HIT
      + /RST_I * CYC * FSTB * /PWC * /PRC * S_DATA;

CYC := /RST_I * /CYC * /FSTB   * BASE_HIT
      + /RST_I * CYC * /FSTB   * S_DATA
      + /RST_I * CYC * FSTB * /PWC   * S_DATA
      + /RST_I * CYC   * /PRC * S_DATA;

CYC_O = CYC;

STB_O = /RST_I   * FSTB
      + /RST_I * /S_WRDN * S_SRC_EN
      + /RST_I * S_WRDN * S_DATA_VLD;
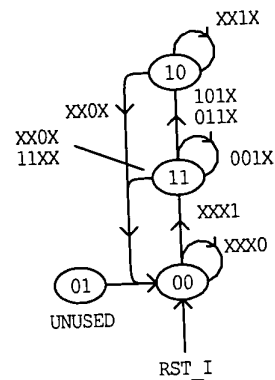
HANDSHAKING STATE MACHINE (PART OF 'CYCLE CONTROL'):

STATES: ABORT, TERM, READY
INPUTS: SDATA, FSTB, ERR_I, RTY_I, ACK_I

EQUATIONS:

S_READY <= READY;
S_TERM <= TERM;
S_ABORT <= ABORT;
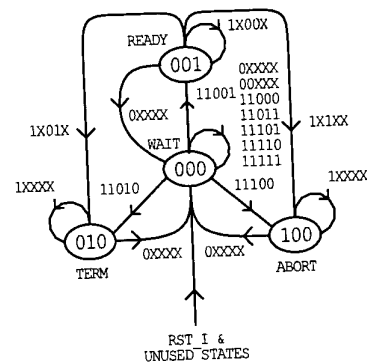
INITIAL CYCLE STATE MACHINE (PART OF 'CYCLE CONTROL'):

STATES: PCYC
INPUTS: BASE_HIT, AER

EQUATIONS:

AER = ACK_I + ERR_I + RTY_I;

PCYC := /RST_I * /PCYC * BASE_HIT
      + /RST_I * PCYC * /AER;

ICYC = PYC * AER;
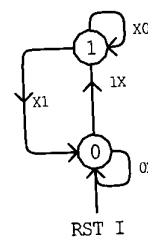
XILINX LOGICORE PCI TO WISHBONE WRAPPER
VHDL ENTITY: PCIWRAP
06 AUG 2002

Figure 4-20. CYCLE CONTROL and HANDSHAKING state machines.

68

4.3.11 Process: LOWER_ADDRESS_COUNTER

The lower address counter generates the lower eight bits of the WISHBONE address bus. The counter is needed because the PCI core generates a base address during the first phase of a bus cycle. Subsequent PCI bus cycles do not generate any address information. The initial base address is preloaded into the counter. After that, each PCI cycle phase increments the counter.

The counter generates the lower eight bits of the local address bus ADR(9..2). The upper twenty-four bits are held by a latch. Local address bus ADR(33..2) generates the WISHBONE address bus ADR_O(33..2). The counter and latch implement the PCI to WISHBONE address translation. The PCI interface uses a 32-bit address bus with four byte enables. This effectively allows 34-bit byte addressability. The WISHBONE interface has an 8-bit granularity. For this reason the PCI address bits are shifted up by two bits, and the byte enables are translated into the WISHBONE SEL_O() bits.

The counter is designed in three sections, with the first two sections having a terminal count (TCNTX) bit. This reduces the number of 'and' terms in the equations of the higher counter bits.

The counter design has been used in other projects, and represents a reasonable compromise between speed and complexity.

Important signals and their uses are:

ADR(9..2):   COUNTER OUTPUT (LOCAL ADDRESS BUS)
ADIO(31..0): COUNTER DATA INPUT (USED FOR PRELOAD)
CINC: COUNTER INCREMENT
ADDR_VLD: COUNTER PRELOAD
TCNT2:   TERMINAL COUNT FROM BIT 2
TCNT5:   TERMINAL COUNT FROM BIT 5

4.3.12 Process: SELECT_CONTROL

The SELECT_CONTROL process generates the local [SEL()] and WISHBONE [SEL_O()] signals.

4.3.13 Process: UPPER_ADDRESS_REGISTER

The UPPER_ADDRESS_REGISTER process creates a 24-bit register, and captures the state of the upper twenty-four address lines coming from ADIO(31..08). These remain static throughout the entire PCI bus cycle, regardless of the type of PCI bus cycle. The lower eight bits are captured by the address counter.

69

Important signals include:

ADR(33..10): REGISTER OUTPUT (LOCAL ADDRESS BUS)
ADIO(31..8): REGISTER DATA INPUT
ADDR_VLD: REGISTER LOAD

4.3.14 Process: WISHBONE_SYSCON

The WISHBONE_SYSCON process generates the WISHBONE SYSCON signals [CLK_O] and [RST_O]. The [CLK_I] and [RST_I] signals should be tied to the [CLK_O] and [RST_O] signals outside of this entity.

[RST_O] is a synchronized version of the asynchronous [RST] signal generated by the Xilinx LogiCORE.

4.3.15 Process: DI_REG

The DI_REG process is a 'data in' register. It's only purpose is to overcome some timing problems associated with the three-state buffers used with the Xilinx LogiCORE PCI. If this register is removed the router attempts to resolve the asynchronous data path from the data in port [DAT_I()], through the three-state buffers, back to the data out port [DAT_O()] and then on to other system components. This timing path is broken when the registers are added.

4.4 SEMABUD Entity

The SEMABUD entity is a shared memory. It is very similar to the SEMABUF entity, except that it implements the memory buffer with Xilinx Spartan 2 distributed RAM instead of Block-Select+ RAM. Furthermore, this entity responds in one clock cycle (the SEMABUF entity responds in two cycles). For more information please see the 'SEMABUF' entity described elsewhere in this manual.

4.5 SEMABUF Entity

The SEMABUF entity is a shared memory buffer. As shown in the block diagram of Figure 4-21, the entity has two WISHBONE SLAVE interfaces called 'PORT A' and 'PORT B'. These connect to two, 256 x 16-bit synchronous memories, thereby forming 32-bit data ports. The memories are created from two Xilinx Spartan 2 BlockSelect+ memories, with details shown in Figure 4-22.

```
                    ┌─────────────────┐
                    │    WISHBONE     │
                    │  SHARED MEMORY  │
                    │     SEMABUF     │
                    │                 │
        ──/──▶──────│ AADR_I()  BADR_I() │◀──/──
           8        │                 │    8
        ──/──▶──────│ ADAT_I()  BDAT_I() │◀──/──
           32       │                 │   32
        ◀──/────────│ ADAT_O()  BDAT_O() │──/──▶
           32       │                 │   32
        ───────▶────│ AWE_I       BWE_I │◀──────
                    │                 │
        ──/──▶──────│ ASEL_I()  BSEL_I() │◀──/──
           2        │                 │    2
        ───────▶────│ ASTB_I     BSTB_I │◀──────
                    │                 │
        ◀───────────│ AACK_O     BACK_O │──────▶
                    │                 │
        ───────▶────│ ACYC_I     BCYC_I │◀──────
                    │                 │
        ───────▶────│ ABRST_I         │
                    │                 │
        ───────▶────│>ABCLK_I         │
                    └─────────────────┘
```
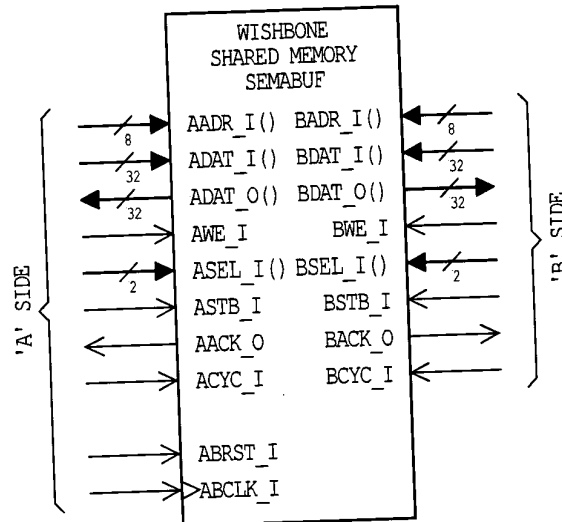
BLOCK DIAGRAM

10 APR, 2002

Figure 4-21. SEMABUF block diagram.

The buffer operates as an independent shared memory. This means that both sides of the memory supports full, simultaneous, read/write privileges into each buffer. During simultaneous transfers one side of the buffer or the other is held off until the other port has finished its operation. In this case the [AACK_O] or [BACK_O] signal holds off memory accesses. This activity is controlled by an internal arbiter circuit.

Table 4-4 gives the specifications for the WISHBONE ports.

An internal arbiter determines whether PORT A or PORT B gains access to the shared memory buffer. The arbiter is composed of a two bit state machine, with a state diagram shown in Figure 4-23, with the related timing diagram shown in Figure 4-24.

72

AACK_O   = ARDY * ASTB_I;

BACK_O   = BGNT * BSTB_I;

XILINX SPARTAN 2 DUAL-PORT SelectRAM+ EQUATIONS

PORTA - BANK #0
WEA   = AWE_I;
ENA   = ARDY̅ * ASTB_I * ASEL_I(0);
RSTA   = GND;
CLKA   = ABCLK_I;
ADDRA(7..0) = ADDR_I̅(8..1);
DINA(15..0) = ADAT̅_I(15..0);
ADAT_O(15..0) = DOA(1̅5..0);

PORTB - BANK #0
WEB   = BWE_I;
ENB   = BGNT̅ * BSTB_I * BSEL_I(0);
RSTB   = GND;
CLKB   = ABCLK_I;
ADDRB(7..0) = BDDR_I̅(8..1);
DINB(15..0) = BDAT̅_I(15..0);
BDAT_O(15..0) = DOB(1̅5..0);

PORTA - BANK #1
WEA   = AWE_I;
ENA   = ARDY̅ * ASTB_I * ASEL_I(1);
RSTA   = GND;
CLKA   = ABCLK_I;
ADDRA(7..0) = ADDR_I̅(8..1);
DINA(15..0) = ADAT̅_I(31..16);
ADAT_O(15..0) = DOA(3̅1..16);

PORTB - BANK #1
WEB   = BWE_I;
ENB   = BGNT̅ * BSTB_I * BSEL_I(1);
RSTB   = GND;
CLKB   = ABCLK_I;
ADDRB(7..0) = BDDR_I̅(8..1);
DINB(15..0) = BDAT̅_I(31..16);
BDAT_O(15..0) = DOB(3̅1..16);

29 SEP 2002

RAMB4_S#_S#

Figure 1: Dual-port Block SelectRAM+ Memory

XILINX SelectRAM+ CONNECTION DIAGRAM

Figure 4-22. Implementation details for the Xilinx BlockSelect+ RAM.

Table 4-4. WISHBONE DATASHEET for the SEMABUF Entity	
General Description	256 x 32-bit shared memory buffer with two WISHBONE SLAVE interfaces. This datasheet applies to both interfaces.
WISHBONE Revision Level	B.2
Supported WISHBONE Cycles	MASTER: READ/WRITE MASTER: BLOCK READ/WRITE
Data port, size	32-bit
Data port, granularity	16-bit
Data port, maximum operand size	32-bit
Data transfer ordering	BIG ENDIAN or LITTLE ENDIAN
Data transfer sequencing	None
Signal Description	All WISHBONE signal names are identical to those defined in the specification, except that they have an 'A' or 'B' at the front. The 'A' and 'B' refer to PORT A and PORTB respectively.
Terminating Signals	The WISHBONE interface on both ports support only the [ACK_O] terminating signal.

SEMABUF ARBITER STATE MACHINE

Figure 4-23. SEMABUF arbiter state machine.

SHARED MEMORY ARBITRATION TIMING

Figure 4-24. SEMABUF shared memory arbiter timing.

4.6 SEMAREG Entity

The SEMAREG entity provides a one-bit semaphore to two WISHBONE SLAVE interfaces. The interfaces are called 'A' and 'B'. The semaphore is intended to be used for shared memory buffers where one interface or the other must gain access to the memory. The semaphore does not lock the memory buffer (which is located elsewhere) but just provides a mechanism for system software to determine if the particular memory buffer is being used. Figure 4-25 shows a block diagram of the entity.

SEMAREG BLOCK DIAGRAM
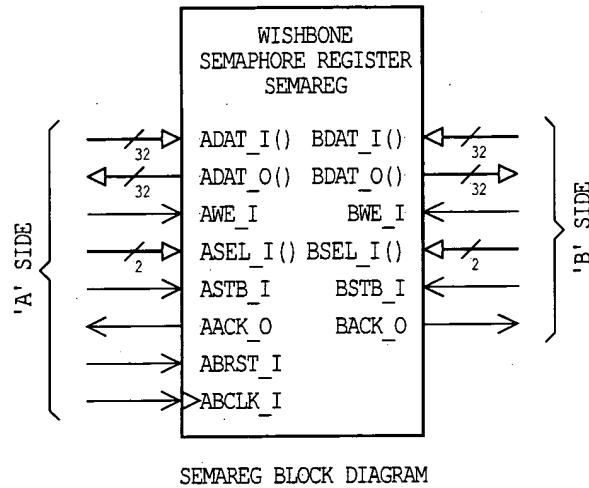
10 APR, 2002

Figure 4-25. SEMAREG block diagram.

The semaphore is accessed by reading the bit. If the bit is returned as '0', then the WISHBONE MASTER (usually a processor) accessing the device is granted the semaphore. If the bit is returned as '1', the buffer is busy. If a processor obtains the buffer by reading '0' (becomes the owner), and the bit is sampled for a second time, then the bit is returned as '1' on the second access.

If both ports attempt to access the semaphore t the same time (i.e. during the same clock cycle), then port 'A' access has priority.

The buffer is released by writing a '0' to the semaphore. Writing a '1' to the bit does not have any effect. The semaphore may be released from either port.

Table 4-5 shows the WISHBONE DATASHEET for the SEMAREG entity.

Table 4-5. WISHBONE DATASHEET for the SEMAREG Entity	
General Description	32-bit semaphore register with WISHBONE SLAVE ports ('A' and 'B'). This datasheet applies to both interfaces.
WISHBONE Revision Level	B.2
Supported WISHBONE Cycles	MASTER: READ/WRITE
Data port, size	32-bit
Data port, granularity	16-bit
Data port, maximum operand size	32-bit
Data transfer ordering	BIG ENDIAN or LITTLE ENDIAN
Data transfer sequencing	None
Signal Description	All WISHBONE signal names are identical to those defined in the specification, except that they have an 'A' or 'B' at the front. The 'A' and 'B' refer to PORT A and PORTB respectively.
Terminating Signals	The WISHBONE interface on both ports support only the [ACK_O] terminating signal.

The circuit arbiter determines whether PORT A or PORT B gains access to the shared memory buffer. The arbiter is composed of a two bit state machine, with a state diagram shown in Figure 4-26. The timing diagram is shown in Figure 4-27.
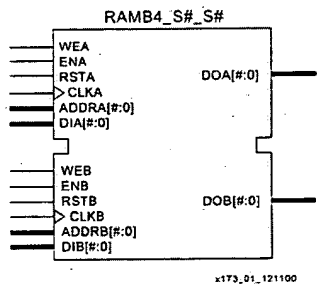
```
STATES: AGNT                    STATES: GNT                     STATES: BGNT
INPUTS: AREQ, GNT               INPUTS: REQ, REL                INPUTS: MBREQ, GNT
CLOCKS: BCLK                    CLOCKS: BCLK                    CLOCKS: BCLK

'A' SIDE PRIORITY                                               'A' SIDE PRIORITY

EQUATIONS:                      EQUATIONS:                      EQUATIONS:

AGNT := /ABRST_I * AREQ * /GNT  GNT := /ABRST_I * /GNT * REQ    BGNT := /ABRST_I * MBREQ * /GNT
      + /ABRST_I * AREQ * AGNT;       + /ABRST_I * GNT * /REL;        + /ABRST_I * MBREQ * BGNT;

                                REQ = AREQ + BREQ;              MBREQ = /AREQ * BREQ;
                                REL = AREL + BREL;
```
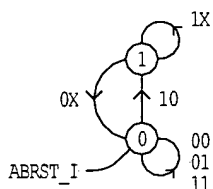


```
       1X                            X0                              1X
OX    10                      X1    1X                        OX    10
         00                            OX                              00
ABRST_I  01                   ABRST_I                         ABRST_I  01
         11                                                            11
```
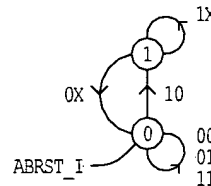
AGNT STATE MACHINE   GNT STATE MACHINE   BGNT STATE MACHINE

OTHER EQUATIONS:

```
AREL     = ASTB_I *  AWE_I * ASEL_I(0) * /ADAT_I(0);
AREQ     = ASTB_I * /AWE_I * ASEL_I(0);
SAACK   := ASTB_I;
AACK_O   = SAACK * ASTB_I;

BREL     = BSTB_I *  BWE_I * BSEL_I(0) * /BDAT_I(0);
BREQ     = BSTB_I * /BWE_I * BSEL_I(0;
SBACK   := BSTB_I;
BACK_O  := SBACK * BSTB_I;
```

30 MAY, 2002

Figure 4-26. SEMAREG state machines.

The semaphore is formed from a series of three, one-bit state machines called 'GNT', 'AGNT' and 'BGNT'. The 'GNT' state machine indicates if the semaphore has been granted or not. Semaphore accesses from either side of the entity will cause the semaphore to be set. The other two state machines determine if one side of the entity or the other is granted the semaphore.

For example, if the semaphore is not granted, and a semaphore request occurs from the 'A' side, then the 'GNT' state machine bit will transition from '0' to '1'. At the same time the 'AGNT' state machine will also transition from '0' to '1'. This grants the semaphore to the 'A' side. Subsequent accesses from the 'A' side will not grant the semaphore, as it is designed so that the semaphore is only granted during the first read access.

Once the semaphore has been granted, neither side can obtain it until the semaphore is released. The release can come from either port of the interface.

79

SEMAPHORE REGISTER (SEMAREG) ARBITRATION TIMING
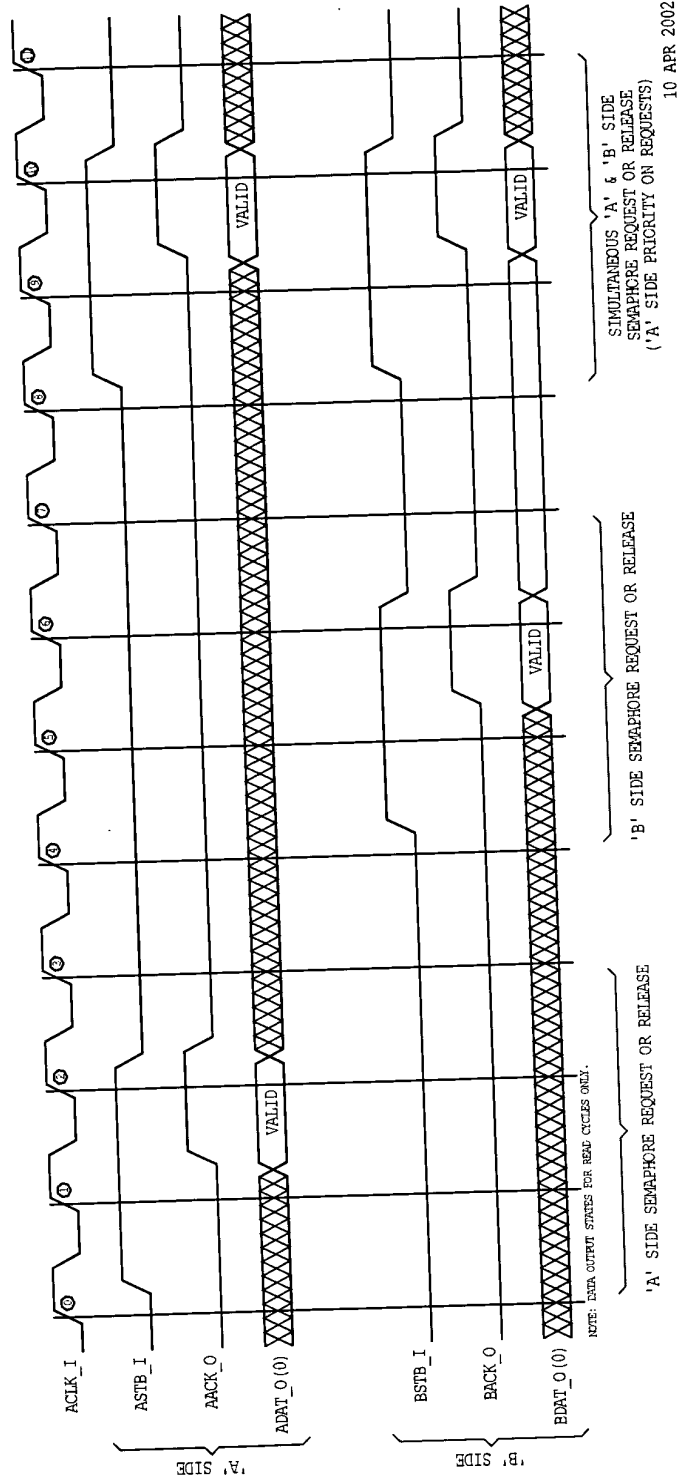
Figure 4-27. Timing diagram for the SEMAREG entity.

4.7 VMEcore™ Entity

The VMEcore™ entity is a A24:D32:D16:D08(O) VMEbus SLAVE to WISHBONE MASTER bridge. As shown in Figure 4-28, the bridge allows connection to a local SoC interconnection. The entire interface is synthesized as the 'VMECOREc' VHDL entity. Table 4-6 shows the characteristics of the WISHBONE interface.

VMEbus / VME64

VMECOREC Entity
VME64 SLAVE | WISHBONE MASTER

WISHBONE

Local System-on-Chip

FPGA or ASIC Target Device

23 SEP 2002
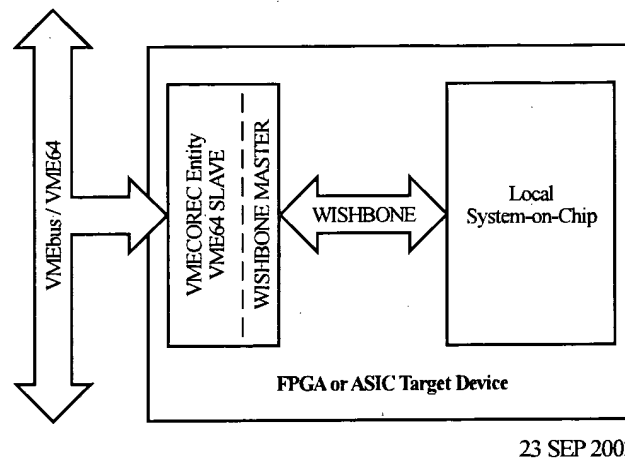
Figure 4-28. Block diagram of the VMEcore™ interface.

Features of the VMEcore(tm) interface include:

o A24:D32:D16:D08(EO) VMEbus slave interface.

o Compact design minimizes gate count and per-unit costs.

o Enforces VMEbus interface rules and timing.

o 32-bit WISHBONE MASTER (backend) interface.

o Allows fabrication of a complete VMEbus SLAVE on a single chip (with peripherals).

o Provided as VHDL source code.

81

Table 4-6. WISHBONE DATASHEET for the VMEcore(tm).	
General Description	Backend interface for a VMEbus SLAVE IP core. Operates as a WISHBONE SoC interface.
WISHBONE Revision Level	B.2
Supported WISHBONE Cycles	MASTER: READ/WRITE MASTER: RMW
Data port, size	32-bit
Data port, granularity	8-bit
Data port, maximum operand size	32-bit
Data transfer ordering	BIG ENDIAN
Data transfer sequencing	UNDEFINED
Signal Description	All WISHBONE signal names are identical to those defined in the specification. Signal [A24SGL_O] is a tag with TAG TYPE: [TGA_O]. Refer to the signal descriptions for more details.
Optional [ERR_I] support	WISHBONE cycles terminated with [ERR_I] terminate VMEbus cycles with BERR*.

4.7.1 VMEbus SLAVE Interface Signals

All VMEbus slave signal names have the '_I' or '_O' characters attached to them. These indicate if the signals are an input (to the core) or an output (from the core). For example, [ACK_I] is an input and [ACK_O] is an output. This convention is used to clearly identify the direction of each signal.

Signal arrays are identified by a name followed by a set of parenthesis. For example, [DAT_I()] is a signal array. Array limits may also be shown within the parenthesis. In this case the first number of the array limit indicates the most significant bit, and the second number indicates the least significant bit. For example, [DAT_I(31..0)] is a signal array with upper array boundary number thirty-one (the most significant bit), and lower array boundary number zero (the least significant bit). The array size on any particular core may vary. In many cases the array boundaries are omitted if they are irrelevant to the context of the description.

When used as part of a sentence, signal names are enclosed in brackets '[ ]'. This helps to discriminate signal names from the words in the sentence.

The VMEbus interface signals can be directly connected to the target device, or routed through buffer chips. Buffer chips are generally used on the VMEbus interface because FPGA and ASIC target devices usually do not have compatible inputs and outputs. VMEbus is based on TTL interface standards, and regulate[24]:

- o Noise margins
- o Load current (inputs)
- o Output short circuit current
- o Input clamp voltage
- o Capacitive loading
- o Output current
- o Backplane impedance

VA_I()
VMEbus address input signal array [VA_I()]. Connect these signals to the corresponding VMEbus address lines A01 – A31 (either directly or through a buffer).

VAM_I()
VMEbus address modifier input signal array [VAM_I()]. Connect these signals to the corresponding VMEbus address modifier lines AM0 – AM5 (either directly or through a buffer).

---

[24] For more information, the reader is directed to the ANSI/VITA 1-1994 standard.

VD_I()
VMEbus data input signal array [VD_I()]. Connect these signals to the corresponding VMEbus data lines D00 – D31 (either directly or through a buffer).

VD_O()
VMEbus data signal output array [VD_O()]. Connect these signals (usually through a buffer) to the corresponding VMEbus data lines D00 – D31. They are generally connected through a buffer to provide sufficient drive current to the VMEbus backplane.

VNAS_I
VMEbus address strobe input signal [VNAS_I]. Connect this signal to the VMEbus [AS*] signal (either directly or through a buffer). Also note that both [VNAS_I] and [AS*] are active low signals.

VNBERR_O
VMEbus bus error output signal [VNBERR_O]. Connect this signal to the VMEbus [BERR*] signal. It is generally connected through a buffer to provide sufficient current drive to the VMEbus backplane. Also note that both [BERR*] and [VNBERR_O] are active low signals.

VNDS0_I
VMEbus data strobe input signal [VNDS0_I]. Connect this signal to the VMEbus [DS0*] signal (either directly or through a buffer). Also note that both [DS0*] and [VNDS0_I] are active low signals.

VNDS1_I
VMEbus data strobe input signal [VNDS1_I]. Connect this signal to the VMEbus [DS1*] signal (either directly or through a buffer). Also note that both [DS1*] and [VNDS1_I] are active low signals.

VNDTACK_O
VMEbus data transfer acknowledge output signal [VNDTACK_O]. Connect this signal to the VMEbus [DTACK*] signal. It is generally connected through a buffer to provide sufficient current drive to the VMEbus backplane. Also note that both [DTACK*] and [VNDTACK_O] are active low signals.

VNIACK_I
VMEbus interrupt acknowledge input signal [VNIACK_I]. Connect this signal to the VMEbus [IACK*] signal (either directly or through a buffer). Also note that both [IACK*] and [VNIACK_I] are active low signals.

VNLWORD_I
VMEbus long word input signal [VNLWORD_I]. Connect this signal to the VMEbus [LWORD*] signal (either directly or through a buffer). Also note that both [LWORD*] and [VNLWORD_I] are active low signals.

VNSYSRESET_I
VMEbus system reset input signal [VNSYSRESET_I]. Connect this signal to the VMEbus [SYSRESET*] signal (either directly or through a buffer). Also note that both [SYSRESET*] and [VNSYSRESET_I] are active low signals.

VNWRITE_I
VMEbus write input signal [VNWRITE_I]. Connect this signal to the VMEbus [WRITE*] signal (either directly or through a buffer). Also note that both [WRITE*] and [VNWRITE_I] are active low signals.

VSAC24_I()
A24 VMEbus SLAVE address compare input signal array [VSAC24_I()]. This array determines when the SLAVE interface is selected by a VMEbus cycle. It is used by the local address comparator to decode the destination address of a bus cycle. They may be connected to a dip-switch or latch on the board which holds the base address of the SLAVE.

VSAE24_I
The A24 VMEbus SLAVE address enable input signal [VSAE24_I] enables the SLAVE interface. In most cases it should be permanently asserted (i.e. tied high). However, in some cases this signal is useful if the interface needs to be disabled.

VTST_I
The simulation test input signal [VTST_I] forces all self-starting counters and other devices in the VMEbus interface to an initial state. It is used for test simulation purposes, and should be negated (i.e. tied low) during normal core operation.

4.7.2 WISHBONE MASTER Interface Signals

A24SGL_O

The [A24SGL_O] signal indicates that a valid A24 VMEbus cycle is in progress, and that the core is participating in the cycle. It conforms to WISHBONE TAG TYPE: [TGA_O]. In many cases this signal is superfluous and can be ignored.

CLK_I

The clock input [CLK_I] coordinates all activities for the internal logic within the WISHBONE interconnect. All WISHBONE output signals are registered at the rising edge of [CLK_I]. All WISHBONE input signals are stable before the rising edge of [CLK_I].

DAT_I()

The data input array [DAT_I()] is used to pass binary data. The array boundaries are determined by the port size, with a maximum port size of 64-bits (e.g. [DAT_I(63..0)]). Also see the [DAT_O()] and [SEL_O()] signal descriptions.

DAT_O()

The data output array [DAT_O()] is used to pass binary data. The array boundaries are determined by the port size, with a maximum port size of 64-bits (e.g. [DAT_I(63..0)]). Also see the [DAT_I()] and [SEL_O()] signal descriptions.

RST_I

The reset input [RST_I] forces the WISHBONE interface to restart. Furthermore, all internal self-starting state machines will be forced into an initial state. This signal only resets the WISHBONE interface. It is not required to reset other parts of an IP core (although it may be used that way).

TGD_I()

Data tag type [TGD_I()] is used on MASTER and SLAVE interfaces. It contains information that is associated with a data lines [DAT_I()], and is qualified by signal [STB_I]. For example, parity protection, error correction and time stamp information can be attached to the data bus. These tag bits simplify the task of defining new signals because their timing (in relation to every bus cycle) is pre-defined by this specification. The name and operation of a data tag must be defined in the WISHBONE DATASHEET.

TGD_O()

Data tag type [TGD_O()] is used on MASTER and SLAVE interfaces. It contains information that is associated with a data lines [DAT_O()], and is qualified by signal [STB_O]. For example, parity protection, error correction and time stamp information can be attached to the data bus. These tag bits simplify the task of defining new signals because their timing (in relation to every

bus cycle) is pre-defined by this specification. The name and operation of a data tag must be defined in the WISHBONE DATASHEET.

ACK_I

The acknowledge input [ACK_I], when asserted, indicates the termination of a normal bus cycle. Also see the [ERR_I] and [RTY_I] signal descriptions.

ADR_O()

The address output array [ADR_O()] is used to pass a binary address. The maximum size of the array is specified as [ADR_O(63..0)]. However, the higher array boundary is specific to the address width of the core, and the lower array boundary is determined by the data port size (e.g. the maximum array size on a 32-bit data port is [ADR_O(63..2)]. In some cases (such as FIFO interfaces) the array may not be present on the interface.

CYC_O

The cycle output [CYC_O], when asserted, indicates that a valid bus cycle is in progress. The signal is asserted for the duration of all bus cycles. For example, during a BLOCK transfer cycle there can be multiple data transfers. The [CYC_O] signal is asserted during the first data transfer, and remains asserted until the last data transfer. The [CYC_O] signal is useful for interfaces with multi-port interfaces (such as dual port memories). In these cases, the [CYC_O] signal requests use of a common bus from an arbiter. Once the arbiter grants the bus to the MASTER, it is held until [CYC_O] is negated.

ERR_I

The error input [ERR_I] indicates an abnormal cycle termination. The source of the error, and the response generated by the MASTER is defined by the IP core supplier. Also see the [ACK_I] and [RTY_I] signal descriptions.

RTY_I

The retry input [RTY_I] indicates that the interface is not ready to accept or send data, and that the cycle should be retried. When and how the cycle is retried is defined by the IP core supplier. Also see the [ERR_I] and [RTY_I] signal descriptions.

SEL_O()

The select output array [SEL_O()] indicates where valid data is expected on the [DAT_I()] signal array during READ cycles, and where it is placed on the [DAT_O()] signal array during WRITE cycles. The array boundaries are determined by the granularity of a port. For example, if 8-bit granularity is used on a 64-bit port, then there would be an array of eight select signals with boundaries of [SEL_O(7..0)]. Each individual select signal correlates to one of eight active bytes on the 64-bit data port. For more information about [SEL_O()], please refer to the data organiza-

tion section in Chapter 3 of this specification. Also see the [DAT_I()], [DAT_O()] and [STB_O]
signal descriptions.

STB_O

The strobe output [STB_O] indicates a valid data transfer cycle. It is used to qualify various
other signals on the interface such as [SEL_O()]. The SLAVE asserts either the [ACK_I],
[ERR_I] or [RTY_I] signals in response to every assertion of the [STB_O] signal.

TGA_O()

Address tag type [TGA_O()] contains information associated with address lines [ADR_O()], and
is qualified by signal [STB_O]. For example, address size (24-bit, 32-bit etc.) and memory man-
agement (protected vs. unprotected) information can be attached to an address. These tag bits
simplify the task of defining new signals because their timing (in relation to every bus cycle) is
defined by this specification. The name and operation of an address tag must be defined in the
WISHBONE DATASHEET.

TGC_O()

Cycle tag type [TGC_O()] contains information associated with bus cycles, and is qualified by
signal [CYC_O]. For example, data transfer, interrupt acknowledge and cache control cycles can
be uniquely identified with the cycle tag. They can also be used to discriminate between
WISHBONE SINGLE, BLOCK and RMW cycles. These tag bits simplify the task of defining
new signals because their timing (in relation to every bus cycle) is defined by this specification.
The name and operation of a cycle tag must be defined in the WISHBONE DATASHEET.

WE_O

The write enable output [WE_O] indicates whether the current local bus cycle is a READ or
WRITE cycle. The signal is negated during READ cycles, and is asserted during WRITE cycles.

4.7.3 VHDL Synthesis and Test

The VMEcore is organized as a series of VHDL entities. These are synthesized together from a
top level entity known as 'VMECOREC.VHD'. Figure 4-29 shows the hierarchy and Figure 4-
30 shows a functional diagram for the core.

Each module of VHDL code is named with a seven character 'handle' that is related to its entity
name. An additional (final) character is added to indicate it's use. The 'C' character indicates
that it's a VHDL circuit file (i.e. entity/architecture pair, usually contained in a file), the 'T'
character indicates that it's a test bench file, and a 'V' character indicates that it's a test vector
file. For example, the top level entity has the following names associated with it:

Entity (circuit) name: VMECOREC
Architecture name: VMECOREC1

Entity/architecture filename: VMECOREC.VHD
Test bench filename: VMECORET.VHD
Text vector filename: VMECOREV.TXT

Only the top level module in the hierarchy includes a VHDL test bench. It is assumed that all of the files are simulated and synthesized as a group.

```
┌─────────────────┐
│  VMECOREC.VHD   │
└─────────────────┘
        │      ┌──────────────────┐
        ├──────│  BUSSCNTC.VHD    │
        │      └──────────────────┘
        │      ┌──────────────────┐
        ├──────│  LOCLADRC.VHD    │
        │      └──────────────────┘
        │      ┌──────────────────┐
        ├──────│  LOCLDATC.VHD    │
        │      └──────────────────┘
        │      ┌──────────────────┐
        ├──────│  SLAVCNTC.VHD    │
        │      └──────────────────┘
        │      ┌──────────────────┐
        ├──────│  SLAVDECC.VHD    │
        │      └──────────────────┘
        │      ┌──────────────────┐
        └──────│  VMEBADBC.VHD    │
               └──────────────────┘
```
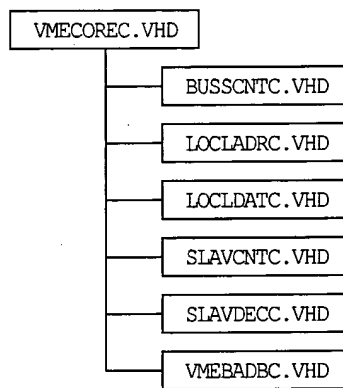
23 SEP 2002

Figure 4-29. VMEcore entities.

The entire set of VMEcore™ entities is created with a parametric core generator called the VMEbus Interface Writer™. The tool itself is a trade secret of Silicore Corporation, and is not available under any license. However, the output files provided as VMEcore entities, test benches and test vector files are fully readable with standard editors. They may be easily and fully modified and maintained without the parametric core generator.
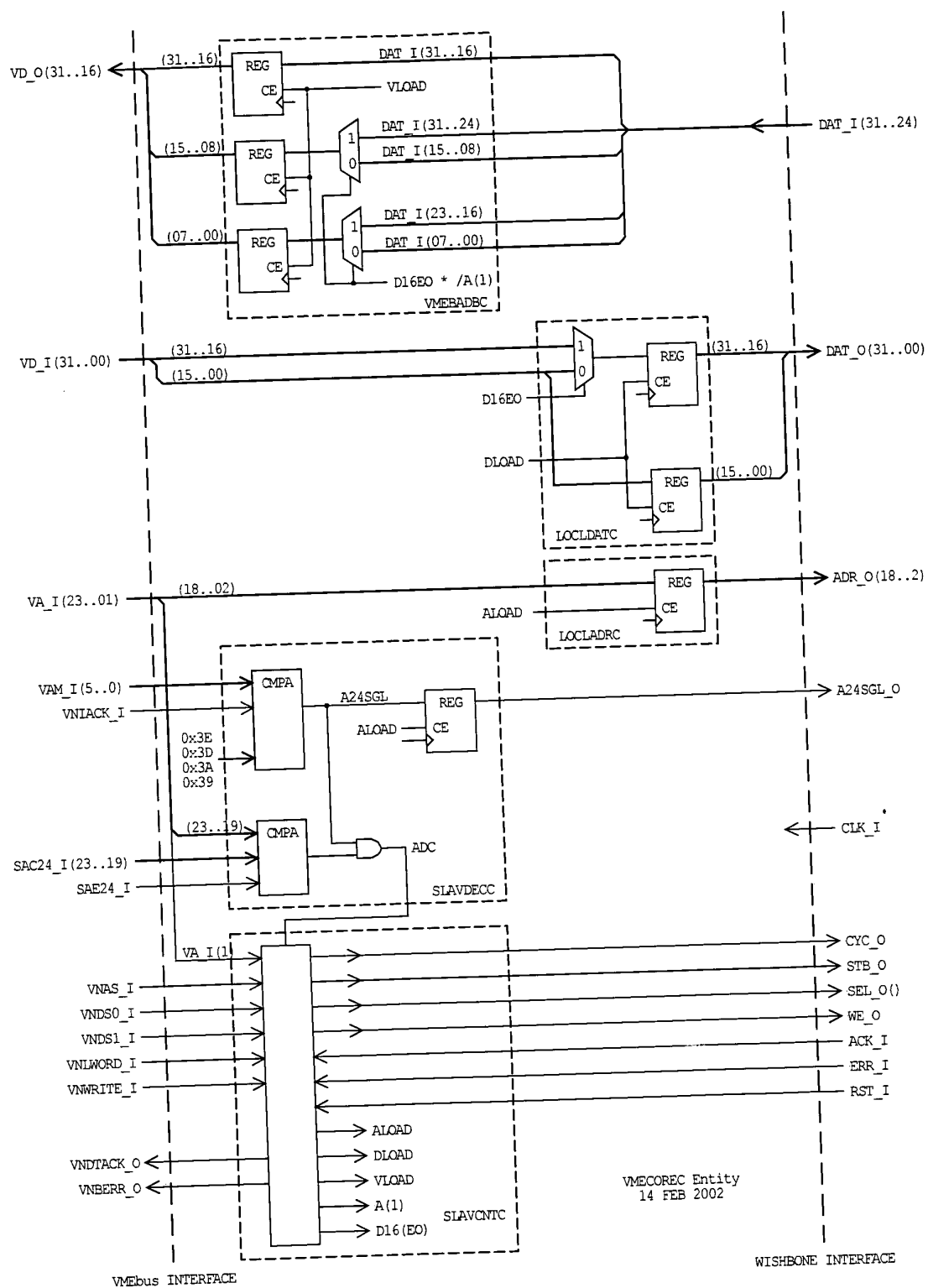
Figure 4-30. Functional diagram of the VMECOREC entity.

4.7.4 Bus Interface Timing

Before routing the VMEcore(tm) interface, correct timing specifications must be entered into the design. Furthermore, these timing specifications must take into account not only the requirements of the VMEcore design, but also of any external driver or receiver chips. These external chips are needed because FPGA and ASIC target devices aren't compatible with the electrical characteristics of the VMEbus backplane.

The VMEcore logical design has only three simple timing constraints:

- Input-to-clock setup time: 1 [VCLK]
- Flop-to-flop transition: 1 [VCLK]
- Clock-to-output delay time: 1 [VCLK]

The *input-to-clock setup time* can be analyzed with Figures 4-31(a) and 4-32(a). This timing parameter includes the time it takes for a VMEbus backplane signal to propagate through an input buffer, through the input pin of the target device and then to set up at the input of a flip-flop.

For example, consider a VMEbus input signal that traverses through an input buffer that has a timing delay [Tplh] of 7.0 ns. Furthermore, assume that [VCLK] operates at a clock speed of 50.000 MHz (or a period of 20.0 ns). This means that the target device must be routed with a worse case input-to-clock delay of:

$$Tpdlhin(max) = Tvclk(min) - Tphl$$

$$Tpdlhin(max) = 20.0 \text{ ns} - 7.0 \text{ ns} = 13.0 \text{ ns}$$

Although these numbers are given for the 'low-to-high' transition delay, a similar case exists for the 'high-to-low' transition delay. The method by which this time specification is entered into the software development tools depends upon the target technology and router used. For example, the timespec for the VMEbus data strobe [N_VDS0] input would be entered into a Xilinx FPGA router thusly:

```
NET "N_VDS0" OFFSET = IN 13 ns BEFORE "VCLK";
```

Tplh
Tphl

Tpdhlin(max)
Tpdlhin(max)

Tvclk(min)

Tpdhlout(max)
Tpdlhout(max)

Tplh
Tphl
Tpoe

VMEbus
Signal

INPUT
BUFFER

D Q

D Q

D Q

Tpoe

OUTPUT
BUFFER

VCLK

Target Device
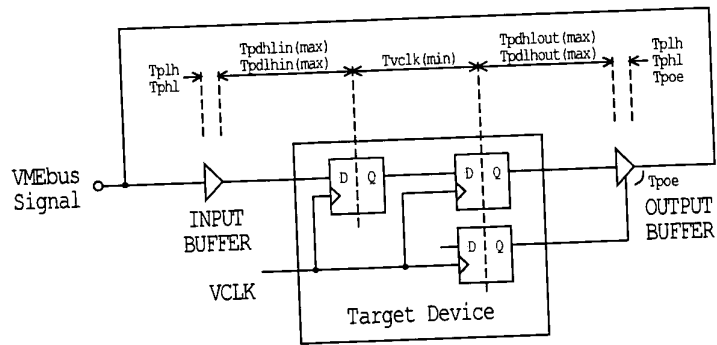
(a) VMEbus Input & Output Signals

Tpdhlin(max)
Tpdlhin(max)

Tvclk(min)

Tpdhlout(max)
Tpdlhout(max)

Tplh
Tphl
Tpoe

VMEbus
Signal

D Q

D Q

D Q

A B

DIR
OE

'1'

BIDIR
BUFFER

VCLK

Target Device
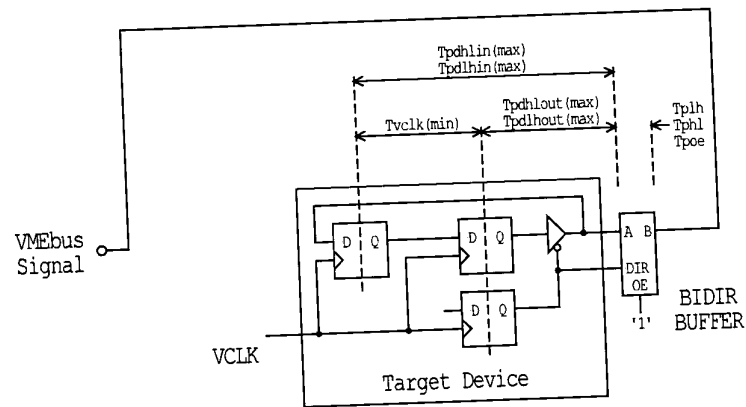
(b) VMEbus Bi-directional Signals

23 SEP 2002

Figure 4-31. Determining bus interface timing characteristics.

92
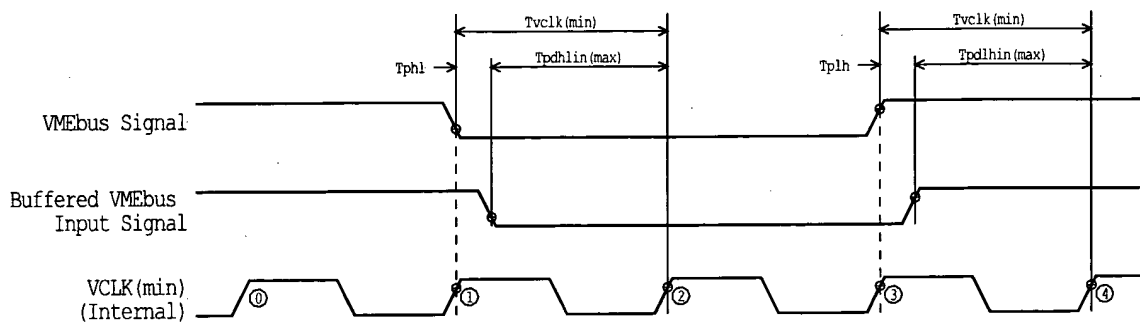
(a) VMEcore Input Timing Constraints

Device input to VCLK setup, high-to-low transition: Tpdhlin(max) = Tvclk(min) - Tphl
Device input to VCLK setup, low-to-high transition: Tpdlhin(max) = Tvclk(min) - Tplh
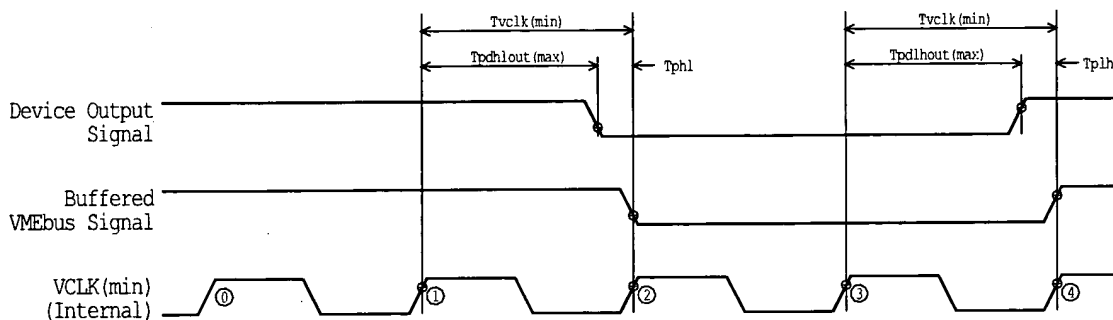
(b) VMEcore Output Timing Constraints

VCLK to device output, high-to-low transition: Tpdhlout(max) = Tvclk(min) - Tphl
VCLK to device output, low-to-high transition: Tpdlhout(max) = Tvclk(min) - Tplh

FEB 08, 2002

Figure 4-32. VMEbus interface timing.

93

The *flop-to-flop transition time* is the time it takes for a synchronous signal (inside the core) to move from the output of one flip-flop and be set-up at the input of another. Stated another way, it is the internal timing of the core's RTL[25] logic. This is shown as [Tvclk(min)] in Figure 4-32(a). For example, if [VCLK] operates at 50.000 MHz, then [Tvclk(min)] is 1/VCLK, or 20.0 ns. It would be entered into a Xilinx FPGA router thusly:

```
NET "VCLK" PERIOD = 30.000;
```

There are additional constraints place on [VCLK] by the VMEbus timing specification. Since VMEbus is asynchronous it must be sampled sufficiently fast to prevent aliasing problems. As shown in Figure 4-33 the core frequency must be at least 33.333 MHz (or a period of 30.0 ns). However, it can't go above 50.000 MHz because the period of [VCLK] must not exceed the data strobe skew on the backplane.
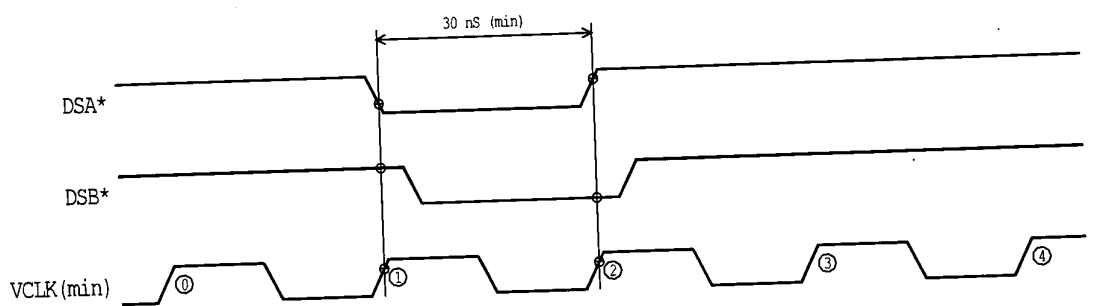
(a) AT VCLK(min) (33.333 MHz) AT LEAST ONE ASSERTED SAMPLE IS ASSURED ON BOTH DATA STROBES.
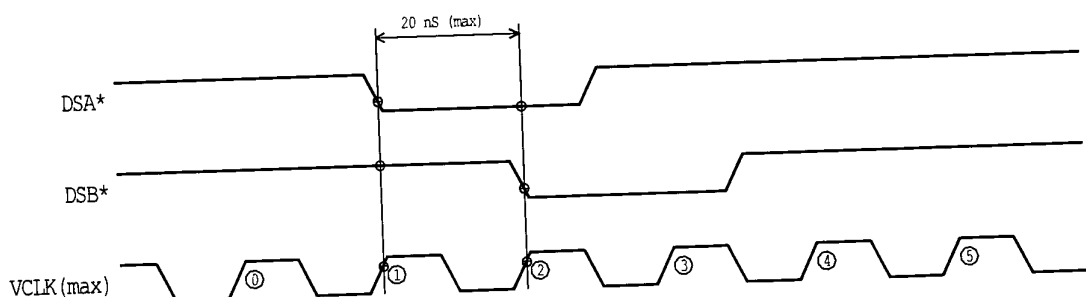
(b) AT VCLK(max) (50.000 MHz) AT LEAST ONE ASSERTED SAMPLE IS ASSURED AT MAXIMUM BUS SKEW.

JAN 24, 2002

Figure 4-33. VMEbus constraints place on [VCLK] frequency.

---

[25] Register-transfer-logic.

94

The *clock-to-output delay time* is the time it takes for a synchronous signal to exit a flip-flop, propagate through the FPGA or ASIC target device, an output buffer, and then arrive at the VMEbus signal interconnection.

For example, consider a VMEbus input signal that traverses through an output buffer that has a timing delay [Tplh] of 10.0 ns. Furthermore, assume that [VCLK] operates at a clock speed of 50.000 MHz (or a period of 20.0 ns). This means that the target device must be routed with a worse case clock-to-output delay of:

$$Tpdlhout(max) = Tvclk(min) - Tphl$$

$$Tpdlhout(max) = 20.0 \text{ ns} - 10.0 \text{ ns} = 10.0 \text{ ns}$$

Although these numbers are given for the 'low-to-high' transition delay, a similar case exists for the 'high-to-low' transition delay. The method by which this time specification is entered into the software development tools depends upon the target technology and router used. For example, the timespec for the VMEbus data line [VD<0>] output would be entered into a Xilinx FPGA router thusly:

```
NET "VD<0>" OFFSET = OUT 10 ns AFTER "VCLK";
```

The timing diagram of Figures 4-34 and 4-35 show how VMEbus read and write cycles are translated to WISHBONE read and write cycles.

VMEcore(tm) TIMING – VMEbus TO WISHBONE SINGLE READ CYCLE
PARTICIPATING VMEbus SLAVE W/NORMAL TERMINATION
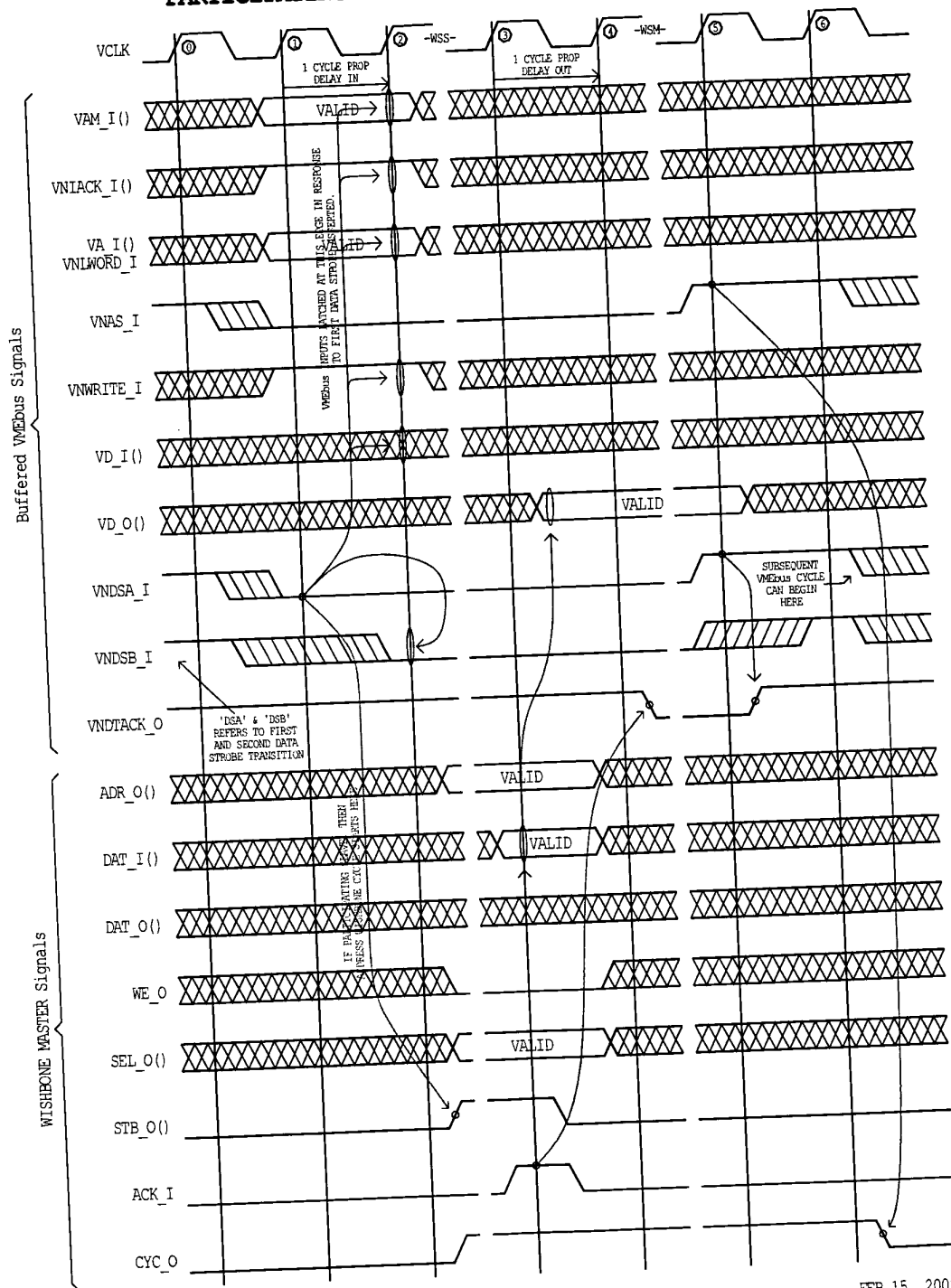
Figure 4-34. VMEbus to WISHBONE read cycle translation.

VMEcore(tm) TIMING - VMEbus TO WISHBONE SINGLE WRITE CYCLE PARTICIPATING VMEbus SLAVE W/NORMAL TERMINATION
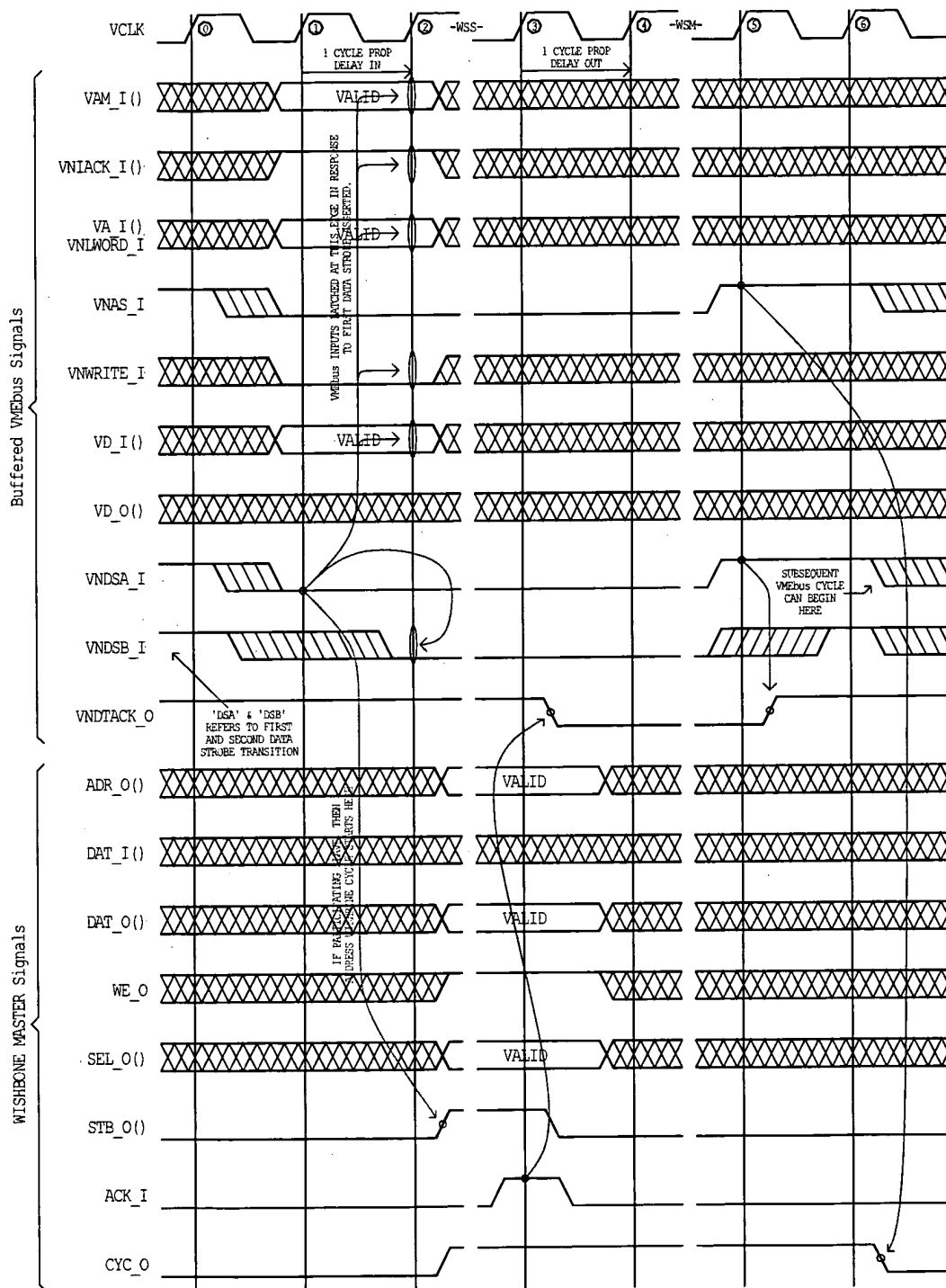
FEB 15, 2002

Figure 4-35. VMEbus to WISHBONE write cycle translation.

4.7.5 BUSSCNTC Entity

The BUSSCNTC (bus controller) entity renames signals used in the core.

4.7.6 LOCLADRC Entity

The LOCLADRC entity latches and routes the VME address bus [VA_I(18..02)] to the WISHBONE address bus [ADR_O(18..02)]. The upper VME address bits [VA_I(23..19)] are only used for address decoding purposes, and are not incorporated into the local SoC address bus. The lower two WISHBONE address bits [ADR_O(01..00)] are not present because they are encoded into select signals [SEL_O(3..0)].

4.7.7 LOCLDATC Entity

The LOCLDATC entity provides two functions: (a) it multiplexes VMEBus data to the correct WISHBONE data bank and (b) it latches the data.

4.7.8 SLAVCNTC Entity

The SLAVCNTC (slave controller) performs miscellaneous control functions for the VMEbus slave interface. This includes VMEbus address and data strobe synchronization, logic sequencer, cycle type generator, slave select logic, slave strobe logic, cycle termination generator (for DTACK*, etc.) and other functions. The SLAVCNTC entity includes the following VHDL processes:

- DCYC process
- DTACK_GENR process
- SEL_GENR process
- SEQU process
- STROBES process
- SYNC process
- TERMINATOR process

The DCYC (data cycle function generator) process identifies the type of participating VMEbus data cycle. During D32 cycles, it asserts signal D32. During D16EO cycles, it asserts D16EO. These signals are used by other processes to correctly route data through the interface.

The DTACK_GENR process is drives the VMEbus DTACK* and BERR* signals. The process also informs other parts of the interface that read data is available the [DREAD] signal.

The [DREAD] signal is asserted when local bus data is available during read cycles. This informs other circuits that data should be latched and presented to the VMEbus data lines.

98

The [VNDTACK_O] signal is connected to DTACK* on the VMEbus interface. During write cycles [VNDTACK_O] is asserted one [CLK_I] edge after WISHBONE acknowledge [ACK_I] is asserted. During read cycles the interface waits for an extra clock cycle before asserting [VNDTACK_O]. This provides extra time to route data through the target device. If the WISHBONE [ERR_I] signal is asserted instead of [ACK_I], then the [VNBERR_O] signal will be asserted instead of [VNDTACK_O]. This indicates that an error occurred during the cycle, and causes the VMEbus [BERR*] signal to be asserted.

The SEL_GENR process drives the WISHBONE MASTER [SEL_O(3..0)] data select signal array. The signal array is used for bank select lines during data transfers. Figure 4-36 shows how the data select signals are asserted during VMEbus transfers.

All of the select signals are asserted during the data load portion of a participating SLAVE cycle. This is indicated by the local [DLOAD] signal, which is generated by the SEQU process. Furthermore, the assertion of an individual signal in the [SEL_O()] array depends on the type of VMEbus data cycle (D32 or D16(EO)) and the address of the transfer. If an individual signal is selected it remains asserted until the cycle is terminated (using [ACK_I], etc.), or when a VMEbus cycle is aborted (indicated by the negation of VMEbus [DS0*] or [DS1*].

VMEcore(tm) WISHBONE MASTER Data Bus Routing - LDSIZE:32									
	VMEbus Signals					SEL_O(3) SEL_O(2) SEL_O(1) SEL_O(0)			
Cycle	IACK*	DS1*	DS0*	LWORD*	A1	DAT_I(31..0) / DAT_O(31..0)			
						31..24	23..16	15..08	07..00
M/S D32	1	0	0	0	0	BYTE(0) D31..D24	BYTE(1) D23..D16	BYTE(2) D15..D08	BYTE(3) D07..D00
MASTER/SLAVE D16(EO)	1	EVEN	ODD	1	1			BYTE(2) D15..D08	BYTE(3) D07..D00
	1	EVEN	ODD	1	0	BYTE(0) D15..D08	BYTE(1) D07..D00		

NOTES: EVEN - DS1* asserted during D16 or even byte transfer.
      ODD - DS0* asserted during D16 or odd byte transfer.
      LW* - VMEbus LWORD* Signal
     (DB) - Carries Data Bit
   SEL_X() - For SLAVE or IREQ: 'X' => 'I'; for MASTER or IHAND: 'X' => '0'.

23 SEP 2002

Figure 4-36. Data select signals during VMEbus transfers.

The SEQU process is a state machine (sequencer) that generates master timing for the core. Figures 4-37 and 4-38 show the state and timing diagrams for the process.

STATE DIAGRAM

CLOCK: CLK I
INPUTS: IHLD, AS, DS
STATES: P2, P1, P0

23 SEP 2002
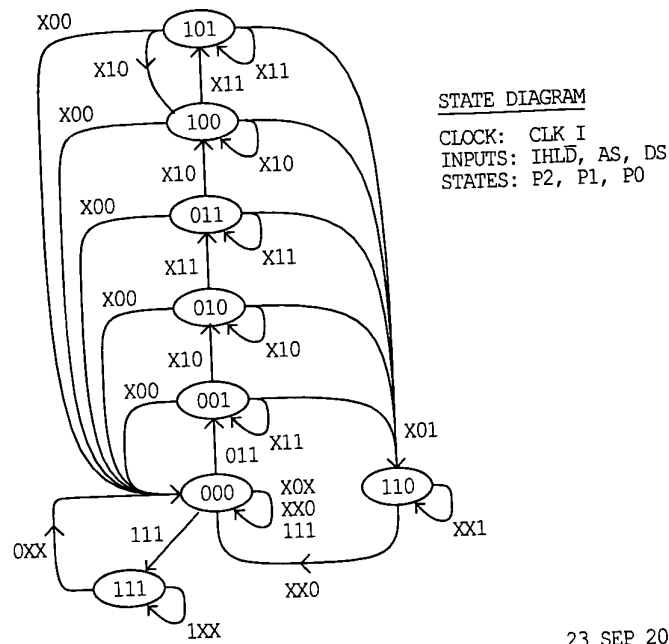
Figure 4-37. State diagram for SEQU process state machine.

ALOAD & DLOAD FUNCTIONS
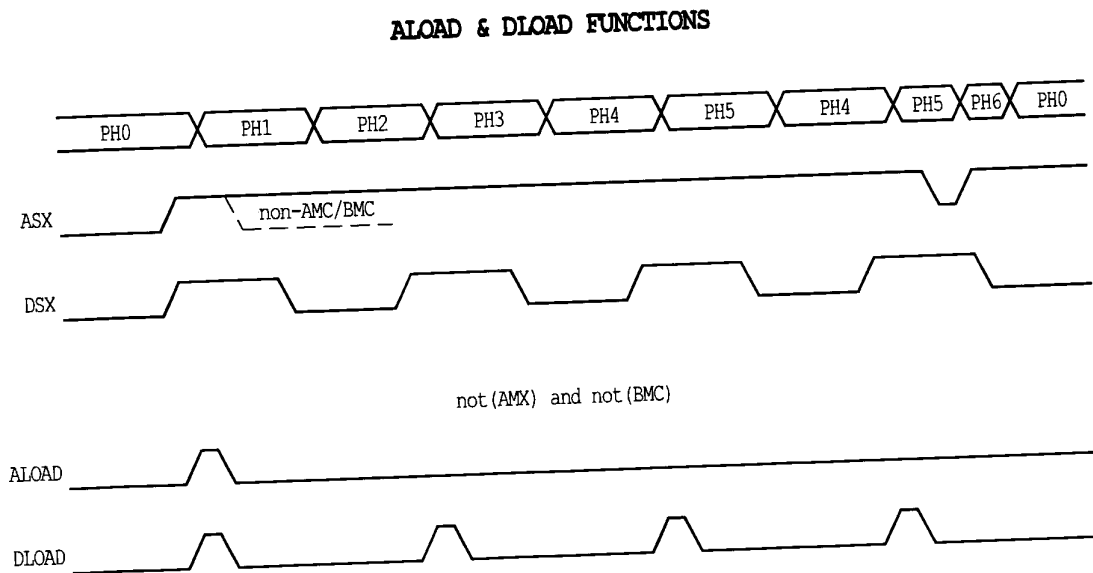
Figure 4-38. Timing diagram (relative) for the SEQU process state machine.

100

The STROBES process drives the WISHBONE [CYC_O], [STB_O] and [WE_O] signals. For more information about these signals please refer to the WISHBONE specification.

The SYNC process synchronizes the VMEbus AS*, DS0* and DS1* signals. It generates local signals [AS], [DS] and [SLDS].

The TERMINATOR process drives a common local signal [ACK] in response to the assertion of [ACK_I] or [ERR_I]. In general, either of these signals can be used to terminate a WISHBONE bus cycle.

4.7.9 SLAVDECC Entity

The SLAVDECC entity decodes the upper five bits of the VMEbus address bus and the address modifier code. When a participating VMEbus interface is selected, the entity asserts the address compare signal [ADC]. The circuit responds to address modifier codes 0x3E, 0x3D, 0x3A and 0x39.

The upper five VMEbus address bits [VA_I(23..19)] are compared to local address compare bits [SAC24_I(23..19)]. When these match (along with the address modifier codes), the interface is selected for a bus cycle.

The interface may be enabled or disabled with signal [SAE24_I]. If the interface is to be permanently enabled, tie [SAE24_I] to logic '1'. Otherwise, it may be used as a generic control signal to enable or disable the interface.

4.7.10 VMEBADBC Entity

The VMEBADBC entity is the data multiplexor and register for the VMEbus data out (VD_O()) signal array. During read cycles, VMEbus output data is routed (using multiplexors) from the WISHBONE data input signal array [DAT_I()] to the correct VMEbus output data signal array [VD_O()]. The routing depends upon the type of VMEbus data transaction (i.e. D32, D16(E) WORD or D16(O) WORD).

4.8 VMEPCIBR Entity

The VMEPCIBR entity is a top level, dual, WISHBONE System-on-Chip (SoC). It is a classic public domain WISHBONE shared bus with multiplexor interconnections that has been modified in several ways. These modifications include:

- Single WISHBONE MASTER operation.
- Dual WISHBONE interconnection buses for VMEbus and PCI sides of the bridge.
- Re-encoded, variable address decoder.

The VMEPCIBR entity is the highest 'RTL' level of the system. A description of the system hierarchy can be found with the VMEPCIBR_SOC description located elsewhere in this manual.

Figure 4-39 is representative of the WISHBONE SoC for both the VMEbus and PCI sides of the bridge. These are called the 'A SIDE' and 'B SIDE' system interconnections, respectively. On the 'A SIDE' interconnection the VMEbus SLAVE (VMEcore) is the WISHBONE MASTER. On the 'B SIDE' interconnection the PCI target (the Xilinx LogiCORE PCI core) is the WISHBONE MASTER. Both sides of the bridge are interconnected through the WISHBONE SLAVEs (i.e. registers and memory buffers).

The WISHBONE variable address decoder was also modified. WISHBONE system address decoders are simplest (and work fastest) when all of the SLAVEs are decoded at locations that are 'powers-of-two' (2, 4, 8, 16, etc.). However, in this design the SLAVES are located at other addresses, so a *re-encoded variable address decoder* is used. Functionally, this is a standard address decoder except that the decoded outputs are again re-encoded to generate evenly spaced select signals for the [ACK_O] and [DRD()] multiplexors. This will slow the system down somewhat, but is necessary in order to generate the correct address map.

Figure 4-40 shows the address decoding for the VMEbus (A_SIDE) WISHBONE interconnection. The PCI side of the bridge has a similar map that is not shown.

Figure 4-41 shows the error decoding for the VMEbus side of the bridge. One requirement of the system is that all non-decoded addresses return an error [ERR_I]. The error decoder output is asserted whenever an error address is selected.

Figure 4-39. Modified public domain WISHBONE shared bus.

103

Figure 4-40. VMEbus (A_SIDE) address decoder.

Destination Entity / File Name	Decode Signal Name	Decode Region – VMEbus BYTE ADDRESSING	High bits	ADDR[17:01]	SEL[3:0]	Data Encoding
MISCREGc.VHD	ADEC00	0x00000	00000	00000000000000000	00	0000
		0x0001F	00000	00000000000001111	11	
		DECODE	00000	0000000000000XXXX	—	
SEMAREGc.VHD 'A'	ADEC01	0x00020	00000	00000000000010000	00	0001
		0x00023	00000	00000000000010001	11	
		DECODE	00000	0000000000001000X	—	
SEMAREGc.VHD 'B'	ADEC02	0x00024	00000	00000000000010010	00	0010
		0x00027	00000	00000000000010011	11	
		DECODE	00000	0000000000001001X	—	
SEMAREGc.VHD 'C'	ADEC03	0x00028	00000	00000000000010100	00	0011
		0x0002B	00000	00000000000010101	11	
		DECODE	00000	0000000000001010X	—	
SEMAREGc.VHD 'D'	ADEC04	0x0002C	00000	00000000000010110	00	0100
		0x0002F	00000	00000000000010111	11	
		DECODE	00000	0000000000001011X	—	
SEMAREGc.VHD 'E'	ADEC05	0x00030	00000	00000000000011000	00	0101
		0x00033	00000	00000000000011001	11	
		DECODE	00000	0000000000001100X	—	
SEMAREGc.VHD 'F'	ADEC06	0x00034	00000	00000000000011010	00	0110
		0x00037	00000	00000000000011011	11	
		DECODE	00000	0000000000001101X	—	
SEMAREGc.VHD 'G'	ADEC07	0x00038	00000	00000000000011100	00	0111
		0x0003B	00000	00000000000011101	11	
		DECODE	00000	0000000000001110X	—	
SEMABUFc.VHD 'A'	ADEC08	0x00800	00000	00000010000000000	00	1000
		0x00BFF	00000	00000010111111111	11	
		DECODE	00000	00000010XXXXXXXXX	—	
SEMABUFc.VHD 'B'	ADEC09	0x00C00	00000	00000011000000000	00	1001
		0x00FFF	00000	00000011111111111	11	
		DECODE	00000	00000011XXXXXXXXX	—	
SEMABUFc.VHD 'C'	ADEC0A	0x01000	00000	00000100000000000	00	1010
		0x013FF	00000	00000100111111111	11	
		DECODE	00000	00000100XXXXXXXXX	—	
SEMABUFc.VHD 'D'	ADEC0B	0x01400	00000	00000101000000000	00	1011
		0x017FF	00000	00000101111111111	11	
		DECODE	00000	00000101XXXXXXXXX	—	
SEMABUFc.VHD 'E'	ADEC0C	0x01800	00000	00000110000000000	00	1100
		0x01BFF	00000	00000110111111111	11	
		DECODE	00000	00000110XXXXXXXXX	—	
SEMABUFc.VHD 'F'	ADEC0D	0x01C00	00000	00000111000000000	00	1101
		0x01FFF	00000	00000111111111111	11	
		DECODE	00000	00000111XXXXXXXXX	—	
SEMABUFc.VHD 'G'	ADEC0E	0x02000	00000	00001000000000000	00	1110
		0x023FF	00000	00001000111111111	11	
		DECODE	00000	00001000XXXXXXXXX	—	
SEMABUFc.VHD 'H'	ADEC0F	0x02400	00000	00001001000000000	00	1111
		0x027FF	00000	00001001111111111	11	
		DECODE	00000	00001001XXXXXXXXX	—	

'A' SIDE ADDRESS DECODING
25 JUL 2002

104

LOCATION NAME	DECODE REGION – VMEbus BYTE ADDRESSING	VA_I(18)	VA_I(17)	VA_I(16)	VA_I(15)	VA_I(14)	VA_I(13)	VA_I(12)	VA_I(11)	VA_I(10)	VA_I(09)	VA_I(08)	VA_I(07)	VA_I(06)	VA_I(05)	VA_I(04)	VA_I(03)	VA_I(02)	SEL(3.0)
		AADR(17)	AADR(16)	AADR(15)	AADR(14)	AADR(13)	AADR(12)	AADR(11)	AADR(10)	AADR(09)	AADR(08)	AADR(07)	AADR(06)	AADR(05)	AADR(04)	AADR(03)	AADR(02)	AADR(01)	
HIGH REGISTERS	0x0003C	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1	1	1	00
	0x0003F	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1	1	1	11
	DECODE	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1	1	1	—
	0x00040	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	00
	0x0007F	0	0	0	0	0	0	0	0	0	0	0	0	1	1	1	1	1	11
	DECODE	0	0	0	0	0	0	0	0	0	0	0	0	1	X	X	X	X	—
	0x00080	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	00
	0x000FF	0	0	0	0	0	0	0	0	0	0	0	1	1	1	1	1	1	11
	DECODE	0	0	0	0	0	0	0	0	0	0	0	1	X	X	X	X	X	—
	0x00100	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	00
	0x001FF	0	0	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	11
	DECODE	0	0	0	0	0	0	0	0	0	0	1	X	X	X	X	X	X	—
	0x00200	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	00
	0x003FF	0	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1	11
	DECODE	0	0	0	0	0	0	0	0	0	1	X	X	X	X	X	X	X	—
	0x00400	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	00
	0x007FF	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1	1	11
	DECODE	0	0	0	0	0	0	0	0	1	X	X	X	X	X	X	X	X	—
HIGH MEMORY	0x02400	0	0	0	0	0	1	0	0	1	0	0	0	0	0	0	0	0	00
	0x027FF	0	0	0	0	0	1	0	0	1	1	1	1	1	1	1	1	1	11
	DECODE	0	0	0	0	0	1	0	0	1	X	X	X	X	X	X	X	X	—
	0x02800	0	0	0	0	0	1	0	1	0	0	0	0	0	0	0	0	0	00
	0x02FFF	0	0	0	0	0	1	0	1	1	1	1	1	1	1	1	1	1	11
	DECODE	0	0	0	0	0	1	0	1	X	X	X	X	X	X	X	X	X	—
	0x03000	0	0	0	0	0	1	1	0	0	0	0	0	0	0	0	0	0	00
	0x03FFF	0	0	0	0	0	1	1	1	1	1	1	1	1	1	1	1	1	11
	DECODE	0	0	0	0	0	1	1	X	X	X	X	X	X	X	X	X	X	—
	0x04000	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	00
	0x07FFF	0	0	0	0	1	1	1	1	1	1	1	1	1	1	1	1	1	11
	DECODE	0	0	0	0	1	X	X	X	X	X	X	X	X	X	X	X	X	—
	0x08000	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	00
	0x0FFFF	0	0	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	11
	DECODE	0	0	0	1	X	X	X	X	X	X	X	X	X	X	X	X	X	—
	0x10000	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	00
	0x1FFFF	0	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	11
	DECODE	0	0	1	X	X	X	X	X	X	X	X	X	X	X	X	X	X	—
	0x20000	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	00
	0x3FFFF	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	11
	DECODE	0	1	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	—
	0x40000	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	00
	0x7FFFF	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	11
	DECODE	1	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	—

Figure 4-41. VMEbus (A_SIDE) error decoder.

4.9 VMEPCIBR_SOC Entity

As shown in Figure 4-42, the highest level entity in the SoC hierarchy is VMEPCIBR_SOC. The entity has two functions: (1) to define the VMEPCIBR entity as a component (the highest level RTL in the SoC) and (2) to specify all of the application specific I/O pads. With the exception of the PCI I/O pads (which Xilinx locates in their PCI core), all I/O pads are located here.
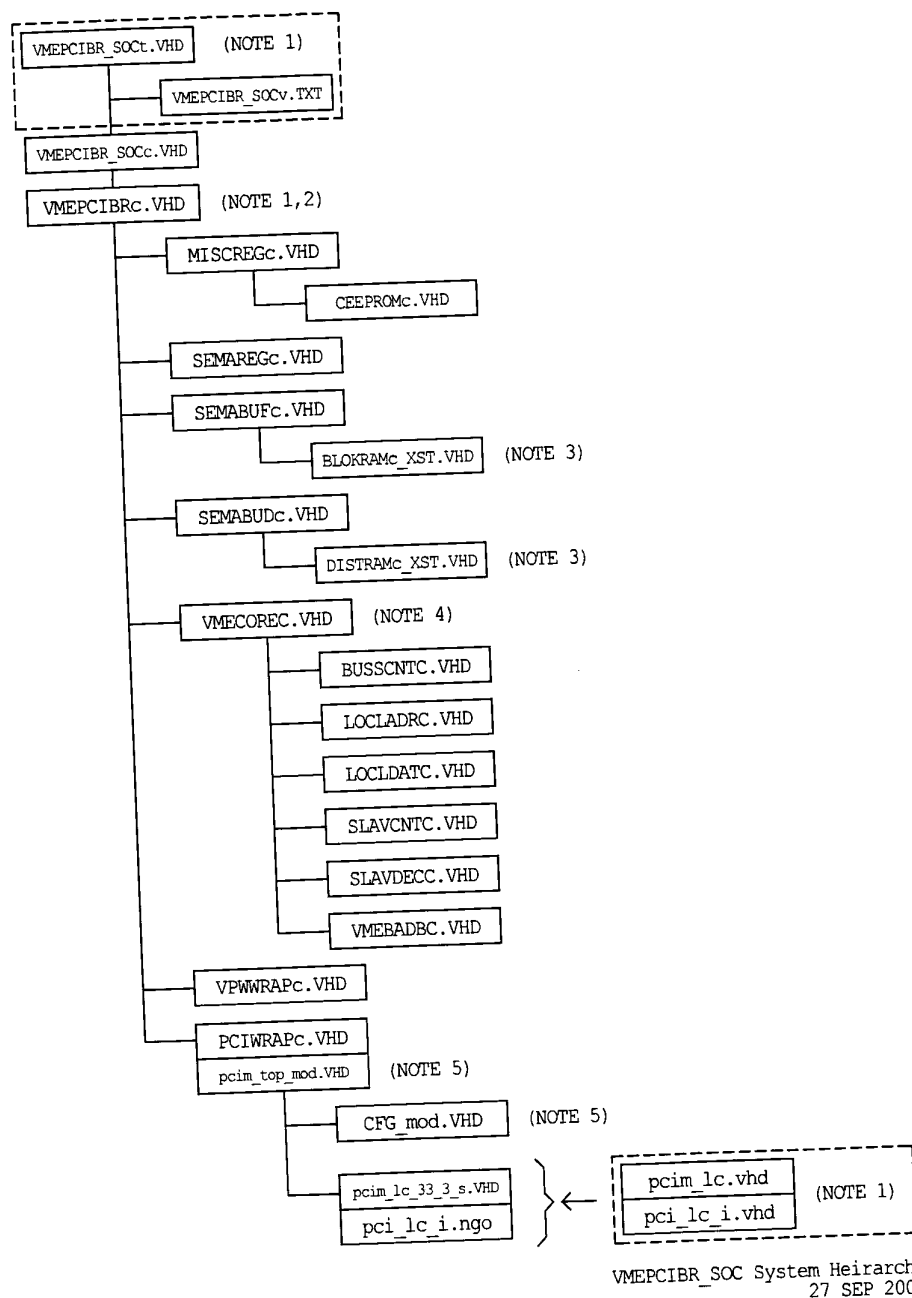
Figure 4-42(a). VMEPCIBR_SOC system hierarchy.

NOTES:

(1) Add or substitute the files shown during pre-synthesis simulation.

The top level test bench named 'VMETOPCI_SOCt.VHD' is used for both the pre-synthesis and post-routing top level simulation. Also note that the test bench uses (reads) a test vector file named 'VMETOPCI_SOCv.TXT'. If the simulator can't find the file, it probably means that its path (as declared in 'VMETOPCI_SOCt.VHD') is wrong.

The unmodified Xilinx PCI wrapper named 'pcim_lc_33_3_s.VHD' is used for synthesis. Substitute file 'pcim_lc.vhd' for pre-synthesis simulation.

The Xilinx PCI core black-box primative named 'pcim_lc_i.ngo' is incorporated into the system by the Xilinx router. Substitute file 'pcim_lc_i.vhd' for pre-synthesis simulation.

Special instructions when using ModelSim:
    (a) After creating the project, be sure to check the "Use 1993 Language Syntax" under Options >> Compile.
    (b) Compile all files from their source directory.

(2) During synthesis, the VHDL source file 'VMETOPCI_SOCc.VHD' is the top level entity.

Route the system with modified Xilinx user constraint file named 2s200pq208_32_33_mod.ucf.

(3) Contains block or distributed RAM primatives generated by Xilinx XST synthesis software.

(4) Silicore VMEcore(tm). Also note that all VMEcore(tm) files are in the VMECORE directory.

(5) Xilinx LogiCORE(tm) PCI file modified for this application. File is located under directory named 'PCIMODS'.

<div align="right">VMEPCIBR_SOC System Heirarchy
27 SEP 2002</div>

Figure 4-42(b). VMEPCIBR_SOC system hierarchy (con't).

4.10 VPWWRAP Entity

VPWWRAP is a VMEbus posted write wrapper for the VMEcore entity. As shown in the functional diagram of Figure 4-43, it contains a register that holds data written from the VMEbus side of the bridge. Immediately after latching the write data, the entity simultaneously: (1) acknowledges the VMEbus cycle by asserting [ACK_O], and (2) begins a WISHBONE write cycle with the latched data. Figure 4-44 and 4-45 shows the timing diagram for read and write cycles (respectively).
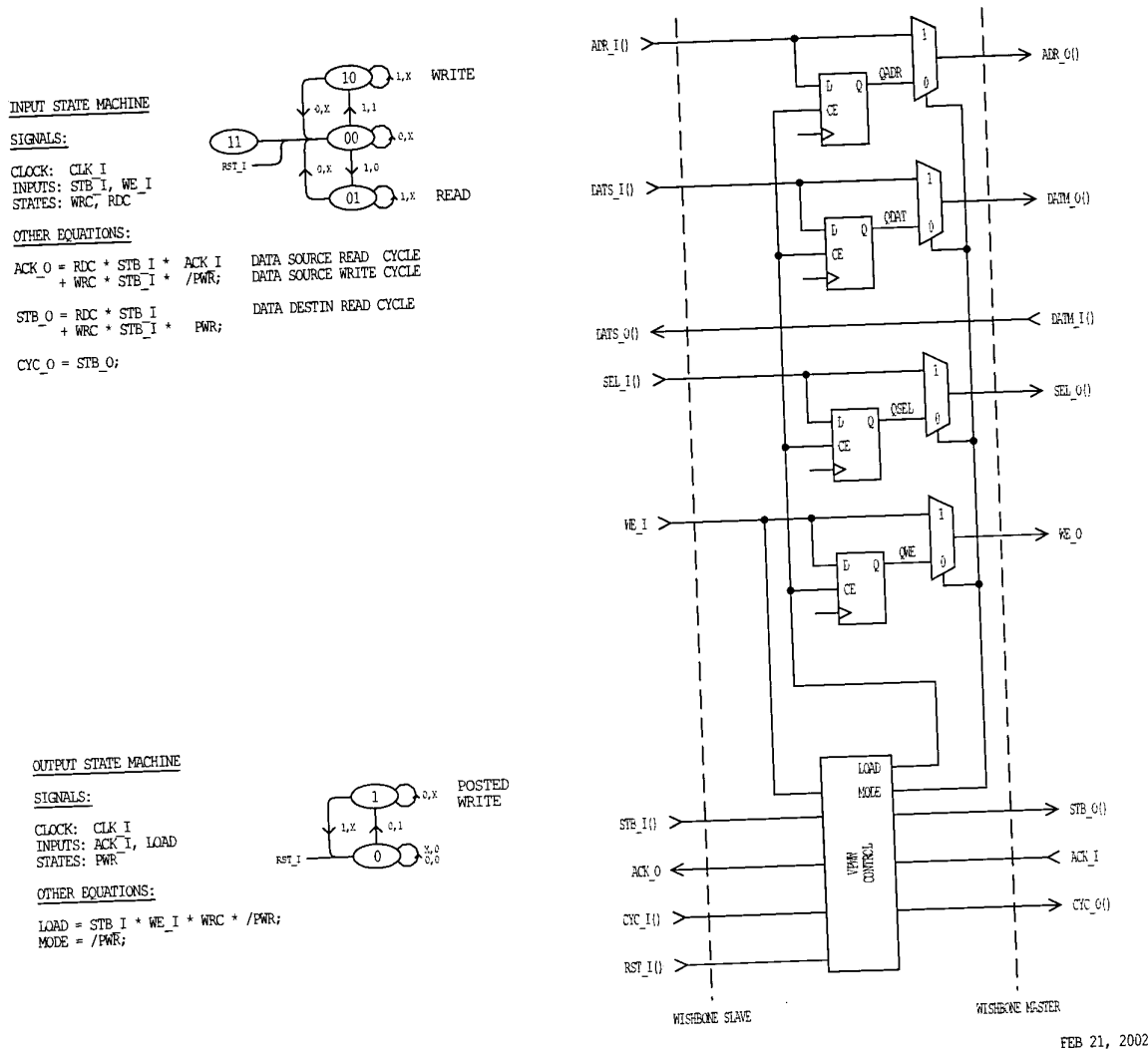
Figure 4-43. Functional diagram of the VPWWRAP entity.

Timing for the VMEcore(tm) Posted Write Wrapper (VPWW)
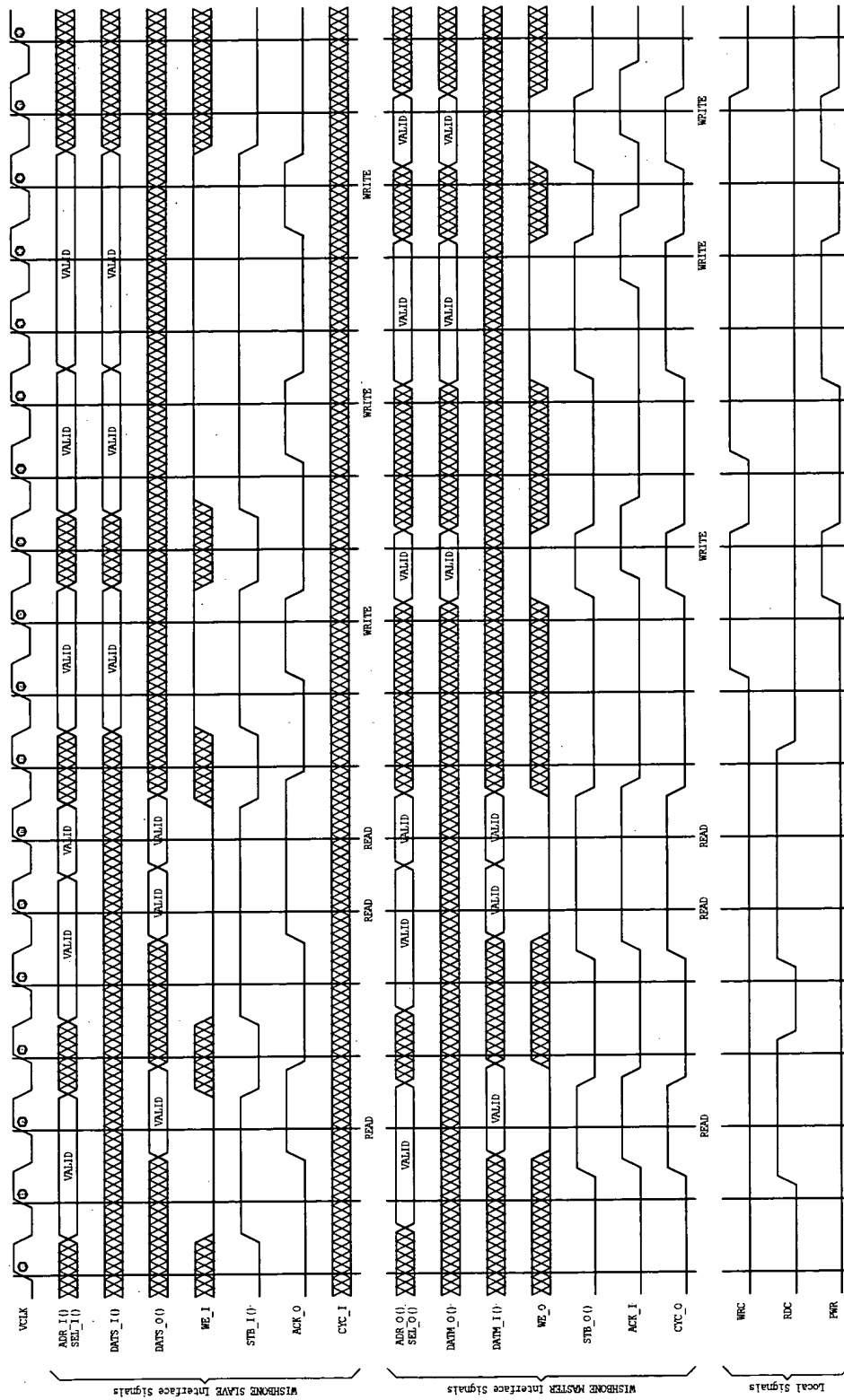
Figure 4-44. VPWWRAP read cycle.
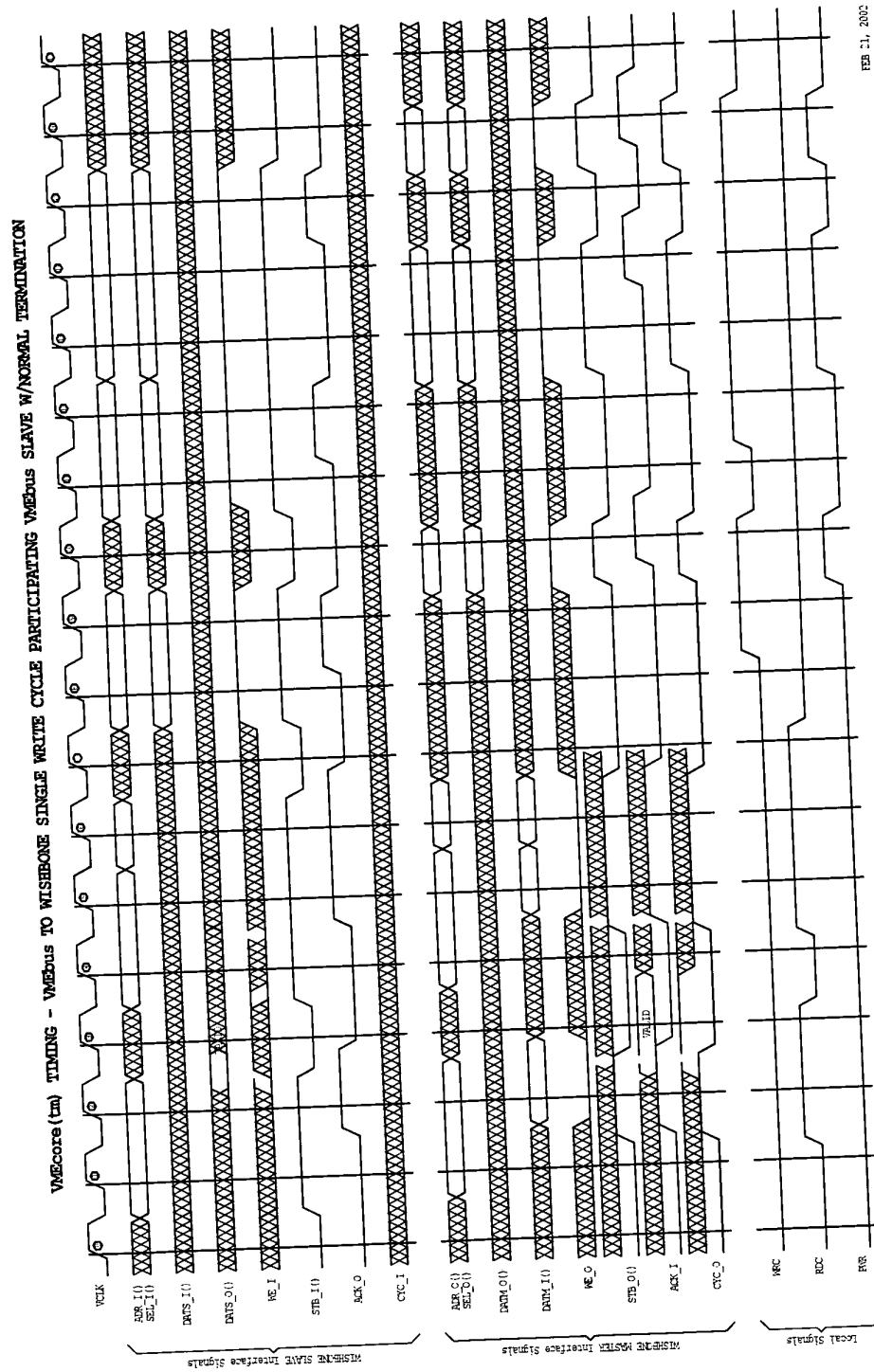
Figure 4-45. VPWWRAP write cycle.

110

Appendix A – GNU LESSER GENERAL PUBLIC LICENSE

GNU LESSER GENERAL PUBLIC LICENSE - Version 2.1, February 1999

Copyright (C) 1991, 1999 Free Software Foundation, Inc. 59 Temple Place, Suite 330, Boston, MA 02111-1307 USA Everyone is permitted to copy and distribute verbatim copies of this license document, but changing it is not allowed.

[This is the first released version of the Lesser GPL. It also counts as the successor of the GNU Library Public License, version 2, hence the version number 2.1.]

PREAMBLE

The licenses for most software are designed to take away your freedom to share and change it. By contrast, the GNU General Public Licenses are intended to guarantee your freedom to share and change free software--to make sure the software is free for all its users.

This license, the Lesser General Public License, applies to some specially designated software packages--typically libraries--of the Free Software Foundation and other authors who decide to use it. You can use it too, but we suggest you first think carefully about whether this license or the ordinary General Public License is the better strategy to use in any particular case, based on the explanations below.

When we speak of free software, we are referring to freedom of use, not price. Our General Public Licenses are designed to make sure that you have the freedom to distribute copies of free software (and charge for this service if you wish); that you receive source code or can get it if you want it; that you can change the software and use pieces of
it in new free programs; and that you are informed that you can do these things.

To protect your rights, we need to make restrictions that forbid distributors to deny you these rights or to ask you to surrender these rights. These restrictions translate to certain responsibilities for you if you distribute copies of the library or if you modify it.

For example, if you distribute copies of the library, whether gratis or for a fee, you must give the recipients all the rights that we gave you. You must make sure that they, too, receive or can get the source code. If you link other code with the library, you must provide complete object files to the recipients, so that they can relink them with the library after making changes to the library and recompiling it. And you must show them these terms so they know their rights.

We protect your rights with a two-step method: (1) we copyright the library, and (2) we offer you this license, which gives you legal permission to copy, distribute and/or modify the library.

To protect each distributor, we want to make it very clear that there is no warranty for the free library. Also, if the library is modified by someone else and passed on, the recipients should

111

know that what they have is not the original version, so that the original author's reputation will not be affected by problems that might be introduced by others.

Finally, software patents pose a constant threat to the existence of any free program. We wish to make sure that a company cannot effectively restrict the users of a free program by obtaining a restrictive license from a patent holder. Therefore, we insist that any patent license obtained for a version of the library must be consistent with the full freedom of use specified in this license.

Most GNU software, including some libraries, is covered by the ordinary GNU General Public License. This license, the GNU Lesser General Public License, applies to certain designated libraries, and is quite different from the ordinary General Public License. We use this license for certain libraries in order to permit linking those libraries into non-free programs.

When a program is linked with a library, whether statically or using a shared library, the combination of the two is legally speaking a combined work, a derivative of the original library. The ordinary General Public License therefore permits such linking only if the entire combination fits its criteria of freedom. The Lesser General Public License permits more lax criteria for linking other code with the library.

We call this license the "Lesser" General Public License because it does Less to protect the user's freedom than the ordinary General Public License. It also provides other free software developers Less of an advantage over competing non-free programs. These disadvantages are the reason we use the ordinary General Public License for many libraries. However, the Lesser license provides advantages in certain special circumstances.

For example, on rare occasions, there may be a special need to encourage the widest possible use of a certain library, so that it becomes a de-facto standard. To achieve this, non-free programs must be allowed to use the library. A more frequent case is that a free library does the same job as widely used non-free libraries. In this case, there is little to gain by limiting the free library to free software only, so we use the Lesser General Public License.

In other cases, permission to use a particular library in non-free programs enables a greater number of people to use a large body of free software. For example, permission to use the GNU C Library in non-free programs enables many more people to use the whole GNU operating system, as well as its variant, the GNU/Linux operating system.

Although the Lesser General Public License is Less protective of the users' freedom, it does ensure that the user of a program that is linked with the Library has the freedom and the wherewithal to run that program using a modified version of the Library.

The precise terms and conditions for copying, distribution and modification follow. Pay close attention to the difference between a "work based on the library" and a "work that uses the library". The former contains code derived from the library, whereas the latter must be combined with the library in order to run.

112

GNU LESSER GENERAL PUBLIC LICENSE TERMS AND CONDITIONS FOR COPYING, DISTRIBUTION AND MODIFICATION

0. This License Agreement applies to any software library or other program which contains a notice placed by the copyright holder or other authorized party saying it may be distributed under the terms of this Lesser General Public License (also called "this License"). Each licensee is addressed as "you".

A "library" means a collection of software functions and/or data prepared so as to be conveniently linked with application programs (which use some of those functions and data) to form executables.

The "Library", below, refers to any such software library or work which has been distributed under these terms. A "work based on the Library" means either the Library or any derivative work under copyright law: that is to say, a work containing the Library or a portion of it, either verbatim or with modifications and/or translated straightforwardly into another language. (Hereinafter, translation is included without limitation in the term "modification".)

"Source code" for a work means the preferred form of the work for making modifications to it. For a library, complete source code means all the source code for all modules it contains, plus any associated interface definition files, plus the scripts used to control compilation and installation of the library.

Activities other than copying, distribution and modification are not covered by this License; they are outside its scope. The act of running a program using the Library is not restricted, and output from such a program is covered only if its contents constitute a work based on the Library (independent of the use of the Library in a tool for writing it). Whether that is true depends on what the Library does and what the program that uses the Library does.

1. You may copy and distribute verbatim copies of the Library's complete source code as you receive it, in any medium, provided that you conspicuously and appropriately publish on each copy an appropriate copyright notice and disclaimer of warranty; keep intact all the notices that refer to this License and to the absence of any warranty; and distribute a copy of this License along with the Library.

You may charge a fee for the physical act of transferring a copy, and you may at your option offer warranty protection in exchange for a fee.

2. You may modify your copy or copies of the Library or any portion of it, thus forming a work based on the Library, and copy and distribute such modifications or work under the terms of Section 1 above, provided that you also meet all of these conditions:

    a) The modified work must itself be a software library.

    b) You must cause the files modified to carry prominent notices stating that you changed the files and the date of any change.

c) You must cause the whole of the work to be licensed at no charge to all third parties under the terms of this License.

d) If a facility in the modified Library refers to a function or a table of data to be supplied by an application program that uses the facility, other than as an argument passed when the facility is invoked, then you must make a good faith effort to ensure that, in the event an application does not supply such function or table, the facility still operates, and performs whatever part of its purpose remains meaningful.

(For example, a function in a library to compute square roots has a purpose that is entirely well-defined independent of the application. Therefore, Subsection 2d requires that any application-supplied function or table used by this function must be optional: if the application does not supply it, the square root function must still compute square roots.)

These requirements apply to the modified work as a whole. If identifiable sections of that work are not derived from the Library, and can be reasonably considered independent and separate works in themselves, then this License, and its terms, do not apply to those sections when you distribute them as separate works. But when you distribute the same sections as part of a whole which is a work based on the Library, the distribution of the whole must be on the terms of this License, whose permissions for other licensees extend to the entire whole, and thus to each and every part regardless of who wrote it.

Thus, it is not the intent of this section to claim rights or contest your rights to work written entirely by you; rather, the intent is to exercise the right to control the distribution of derivative or collective works based on the Library.

In addition, mere aggregation of another work not based on the Library with the Library (or with a work based on the Library) on a volume of a storage or distribution medium does not bring the other work under the scope of this License.

3. You may opt to apply the terms of the ordinary GNU General Public License instead of this License to a given copy of the Library. To do this, you must alter all the notices that refer to this License, so that they refer to the ordinary GNU General Public License, version 2, instead of to this License. (If a newer version than version 2 of the ordinary GNU General Public License has appeared, then you can specify that version instead if you wish.) Do not make any other change in these notices.

Once this change is made in a given copy, it is irreversible for that copy, so the ordinary GNU General Public License applies to all subsequent copies and derivative works made from that copy.

This option is useful when you wish to copy part of the code of the Library into a program that is not a library.

4. You may copy and distribute the Library (or a portion or derivative of it, under Section 2) in object code or executable form under the terms of Sections 1 and 2 above provided that you accompany it with the complete corresponding machine-readable source code, which must be distributed under the terms of Sections 1 and 2 above on a medium customarily used for software interchange.

If distribution of object code is made by offering access to copy from a designated place, then offering equivalent access to copy the source code from the same place satisfies the requirement to distribute the source code, even though third parties are not compelled to copy the source along with the object code.

5. A program that contains no derivative of any portion of the Library, but is designed to work with the Library by being compiled or linked with it, is called a "work that uses the Library". Such a work, in isolation, is not a derivative work of the Library, and therefore falls outside the scope of this License.

However, linking a "work that uses the Library" with the Library creates an executable that is a derivative of the Library (because it contains portions of the Library), rather than a "work that uses the library". The executable is therefore covered by this License. Section 6 states terms for distribution of such executables.

When a "work that uses the Library" uses material from a header file that is part of the Library, the object code for the work may be a derivative work of the Library even though the source code is not. Whether this is true is especially significant if the work can be linked without the Library, or if the work is itself a library. The threshold for this to be true is not precisely defined by law.

If such an object file uses only numerical parameters, data structure layouts and accessors, and small macros and small inline functions (ten lines or less in length), then the use of the object file is unrestricted, regardless of whether it is legally a derivative work. (Executables containing this object code plus portions of the Library will still fall under Section 6.)

Otherwise, if the work is a derivative of the Library, you may distribute the object code for the work under the terms of Section 6. Any executables containing that work also fall under Section 6, whether or not they are linked directly with the Library itself.

6. As an exception to the Sections above, you may also combine or link a "work that uses the Library" with the Library to produce a work containing portions of the Library, and distribute that work under terms of your choice, provided that the terms permit modification of the work for the customer's own use and reverse engineering for debugging such modifications.

You must give prominent notice with each copy of the work that the Library is used in it and that the Library and its use are covered by this License. You must supply a copy of this License. If the work during execution displays copyright notices, you must include the copyright notice for

the Library among them, as well as a reference directing the user to the copy of this License. Also, you must do one of these things:

a) Accompany the work with the complete corresponding machine-readable source code for the Library including whatever changes were used in the work (which must be distributed under Sections 1 and 2 above); and, if the work is an executable linked with the Library, with the complete machine-readable "work that uses the Library", as object code and/or source code, so that the user can modify the Library and then relink to produce a modified executable containing the modified Library. (It is understood that the user who changes the contents of definitions files in the Library will not necessarily be able to recompile the application to use the modified definitions.)

b) Use a suitable shared library mechanism for linking with the Library. A suitable mechanism is one that (1) uses at run time a copy of the library already present on the user's computer system, rather than copying library functions into the executable, and (2) will operate properly with a modified version of the library, if the user installs one, as long as the modified version is interface-compatible with the version that the work was made with.

c) Accompany the work with a written offer, valid for at least three years, to give the same user the materials specified in Subsection 6a, above, for a charge no more than the cost of performing this distribution.

d) If distribution of the work is made by offering access to copy from a designated place, offer equivalent access to copy the above specified materials from the same place.

e) Verify that the user has already received a copy of these materials or that you have already sent this user a copy.

For an executable, the required form of the "work that uses the Library" must include any data and utility programs needed for reproducing the executable from it. However, as a special exception, the materials to be distributed need not include anything that is normally distributed (in either source or binary form) with the major components (compiler, kernel, and so on) of the operating system on which the executable runs, unless that component itself accompanies the executable.

It may happen that this requirement contradicts the license restrictions of other proprietary libraries that do not normally accompany the operating system. Such a contradiction means you cannot use both them and the Library together in an executable that you distribute.

7. You may place library facilities that are a work based on the Library side-by-side in a single library together with other library facilities not covered by this License, and distribute such a combined library, provided that the separate distribution of the work based on the Library and of the other library facilities is otherwise permitted, and provided that you do these two things:

a) Accompany the combined library with a copy of the same work based on the Library, uncombined with any other library facilities. This must be distributed under the terms of the Sections above.

b) Give prominent notice with the combined library of the fact that part of it is a work based on the Library, and explaining where to find the accompanying uncombined form of the same work.

8. You may not copy, modify, sublicense, link with, or distribute the Library except as expressly provided under this License. Any attempt otherwise to copy, modify, sublicense, link with, or distribute the Library is void, and will automatically terminate your rights under this License. However, parties who have received copies, or rights, from you under this License will not have their licenses terminated so long as such parties remain in full compliance.

9. You are not required to accept this License, since you have not signed it. However, nothing else grants you permission to modify or distribute the Library or its derivative works. These actions are prohibited by law if you do not accept this License. Therefore, by modifying or distributing the Library (or any work based on the Library), you indicate your acceptance of this License to do so, and all its terms and conditions for copying, distributing or modifying the Library or works based on it.

10. Each time you redistribute the Library (or any work based on the Library), the recipient automatically receives a license from the original licensor to copy, distribute, link with or modify the Library subject to these terms and conditions. You may not impose any further restrictions on the recipients' exercise of the rights granted herein. You are not responsible for enforcing compliance by third parties with this License.

11. If, as a consequence of a court judgment or allegation of patent infringement or for any other reason (not limited to patent issues), conditions are imposed on you (whether by court order, agreement or otherwise) that contradict the conditions of this License, they do not excuse you from the conditions of this License. If you cannot distribute so as to satisfy simultaneously your obligations under this License and any other pertinent obligations, then as a consequence you may not distribute the Library at all. For example, if a patent license would not permit royalty-free redistribution of the Library by all those who receive copies directly or indirectly through you, then the only way you could satisfy both it and this License would be to refrain entirely from distribution of the Library.

If any portion of this section is held invalid or unenforceable under any particular circumstance, the balance of the section is intended to apply, and the section as a whole is intended to apply in other circumstances.

It is not the purpose of this section to induce you to infringe any patents or other property right claims or to contest validity of any such claims; this section has the sole purpose of protecting the integrity of the free software distribution system which is implemented by public license practices. Many people have made generous contributions to the wide range of software distributed through that system in reliance on consistent application of that system; it is up to the au-

thor/donor to decide if he or she is willing to distribute software through any other system and a licensee cannot impose that choice.

This section is intended to make thoroughly clear what is believed to be a consequence of the rest of this License.

12. If the distribution and/or use of the Library is restricted in certain countries either by patents or by copyrighted interfaces, the original copyright holder who places the Library under this License may add an explicit geographical distribution limitation excluding those countries, so that distribution is permitted only in or among countries not thus excluded. In such case, this License incorporates the limitation as if written in the body of this License.

13. The Free Software Foundation may publish revised and/or new versions of the Lesser General Public License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns.

Each version is given a distinguishing version number. If the Library specifies a version number of this License which applies to it and "any later version", you have the option of following the terms and conditions either of that version or of any later version published by the Free Software Foundation. If the Library does not specify a license version number, you may choose any version ever published by the Free Software Foundation.

14. If you wish to incorporate parts of the Library into other free programs whose distribution conditions are incompatible with these, write to the author to ask for permission. For software which is copyrighted by the Free Software Foundation, write to the Free Software Foundation; we sometimes make exceptions for this. Our decision will be guided by the two goals of preserving the free status of all derivatives of our free software and of promoting the sharing and reuse of software generally.

NO WARRANTY

15. BECAUSE THE LIBRARY IS LICENSED FREE OF CHARGE, THERE IS NO WARRANTY FOR THE LIBRARY, TO THE EXTENT PERMITTED BY APPLICABLE LAW. EXCEPT WHEN OTHERWISE STATED IN WRITING THE COPYRIGHT HOLDERS AND/OR OTHER PARTIES PROVIDE THE LIBRARY "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESSED OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. THE ENTIRE RISK AS TO THE QUALITY AND PERFORMANCE OF THE LIBRARY IS WITH YOU. SHOULD THE LIBRARY PROVE DEFECTIVE, YOU ASSUME THE COST OF ALL NECESSARY SERVICING, REPAIR OR CORRECTION.

16. IN NO EVENT UNLESS REQUIRED BY APPLICABLE LAW OR AGREED TO IN WRITING WILL ANY COPYRIGHT HOLDER, OR ANY OTHER PARTY WHO MAY MODIFY AND/OR REDISTRIBUTE THE LIBRARY AS PERMITTED ABOVE, BE LIABLE TO YOU FOR DAMAGES, INCLUDING ANY GENERAL, SPECIAL, INCIDENTAL OR CONSEQUENTIAL DAMAGES ARISING OUT OF THE USE OR INABILITY TO USE THE

LIBRARY (INCLUDING BUT NOT LIMITED TO LOSS OF DATA OR DATA BEING RENDERED INACCURATE OR LOSSES SUSTAINED BY YOU OR THIRD PARTIES OR A FAILURE OF THE LIBRARY TO OPERATE WITH ANY OTHER SOFTWARE), EVEN IF SUCH HOLDER OR OTHER PARTY HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

END OF TERMS AND CONDITIONS

HOW TO APPLY THESE TERMS TO YOUR NEW LIBRARIES

If you develop a new library, and you want it to be of the greatest possible use to the public, we recommend making it free software that everyone can redistribute and change. You can do so by permitting redistribution under these terms (or, alternatively, under the terms of the ordinary General Public License).

To apply these terms, attach the following notices to the library. It is safest to attach them to the start of each source file to most effectively convey the exclusion of warranty; and each file should have at least the "copyright" line and a pointer to where the full notice is found.

> <one line to give the library's name and a brief idea of what it does.>
> Copyright (C) <year> <name of author>
>
> This library is free software; you can redistribute it and/or modify it under the terms of the GNU Lesser General Public License as published by the Free Software Foundation; either version 2.1 of the License, or (at your option) any later version.
>
> This library is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU Lesser General Public License for more details.
>
> You should have received a copy of the GNU Lesser General Public License along with this library; if not, write to the Free Software Foundation, Inc., 59 Temple Place, Suite 330, Boston, MA 02111-1307 USA
>
> Also add information on how to contact you by electronic and paper mail.

You should also get your employer (if you work as a programmer) or your school, if any, to sign a "copyright disclaimer" for the library, ifnecessary. Here is a sample; alter the names:

Yoyodyne, Inc., hereby disclaims all copyright interest in the library `Frob' (a library for tweaking knobs) written by James Random Hacker.
<signature of Ty Coon>, 1 April 1990
Ty Coon, President of Vice

That's all there is to it!

Appendix C – GNU Free Documentation License

GNU Free Documentation License - Version 1.2, November 2002

0. PREAMBLE

The purpose of this License is to make a manual, textbook, or other functional and useful document "free" in the sense of freedom: to assure everyone the effective freedom to copy and redistribute it, with or without modifying it, either commercially or non-commercially. Secondarily, this License preserves for the author and publisher a way to get credit for their work, while not being considered responsible for modifications made by others.

This License is a kind of "copyleft", which means that derivative works of the document must themselves be free in the same sense. It complements the GNU General Public License, which is a copyleft license designed for free software.

We have designed this License in order to use it for manuals for free software, because free software needs free documentation: a free program should come with manuals providing the same freedoms that the software does. But this License is not limited to software manuals; it can be used for any textual work, regardless of subject matter or whether it is published as a printed book. We recommend this License principally for works whose purpose is instruction or reference.

1. APPLICABILITY AND DEFINITIONS

This License applies to any manual or other work, in any medium, that contains a notice placed by the copyright holder saying it can be distributed under the terms of this License. Such a notice grants a world-wide, royalty-free license, unlimited in duration, to use that work under the conditions stated herein. The "Document", below, refers to any such manual or work. Any member of the public is a licensee, and is addressed as "you". You accept the license if you copy, modify or distribute the work in a way requiring permission under copyright law.

A "Modified Version" of the Document means any work containing the Document or a portion of it, either copied verbatim, or with modifications and/or translated into another language.

A "Secondary Section" is a named appendix or a front-matter section of the Document that deals exclusively with the relationship of the publishers or authors of the Document to the Document's overall subject (or to related matters) and contains nothing that could fall directly within that overall subject. (Thus, if the Document is in part a textbook of mathematics, a Secondary Section may not explain any mathematics.) The relationship could be a matter of historical connec-

tion with the subject or with related matters, or of legal, commercial, philosophical, ethical or political position regarding them.

The "Invariant Sections" are certain Secondary Sections whose titles are designated, as being those of Invariant Sections, in the notice that says that the Document is released under this License. If a section does not fit the above definition of Secondary then it is not allowed to be designated as Invariant. The Document may contain zero Invariant Sections. If the Document does not identify any Invariant Sections then there are none.

The "Cover Texts" are certain short passages of text that are listed, as Front-Cover Texts or Back-Cover Texts, in the notice that says that the Document is released under this License. A Front-Cover Text may be at most 5 words, and a Back-Cover Text may be at most 25 words.

A "Transparent" copy of the Document means a machine-readable copy, represented in a format whose specification is available to the general public, that is suitable for revising the document straightforwardly with generic text editors or (for images composed of pixels) generic paint programs or (for drawings) some widely available drawing editor, and that is suitable for input to text formatters or for automatic translation to a variety of formats suitable for input to text formatters. A copy made in an otherwise Transparent file format whose markup, or absence of markup, has been arranged to thwart or discourage subsequent modification by readers is not Transparent. An image format is not Transparent if used for any substantial amount of text. A copy that is not "Transparent" is called "Opaque".

Examples of suitable formats for Transparent copies include plain ASCII without markup, Texinfo input format, LaTeX input format, SGML or XML using a publicly available DTD, and standard-conforming simple HTML, PostScript or PDF designed for human modification. Examples of transparent image formats include PNG, XCF and JPG. Opaque formats include proprietary formats that can be read and edited only by proprietary word processors, SGML or XML for which the DTD and/or processing tools are not generally available, and the machine-generated HTML, PostScript or PDF produced by some word processors for output purposes only.

The "Title Page" means, for a printed book, the title page itself, plus such following pages as are needed to hold, legibly, the material this License requires to appear in the title page. For works in formats which do not have any title page as such, "Title Page" means the text near the most prominent appearance of the work's title, preceding the beginning of the body of the text.

A section "Entitled XYZ" means a named subunit of the Document whose title either is precisely XYZ or contains XYZ in parentheses following text that translates XYZ in another language. (Here XYZ stands for a specific section name mentioned below, such as "Acknowledgements", "Dedications", "Endorsements", or "History".) To "Preserve the Title" of such a section when you modify the Document means that it remains a section "Entitled XYZ" according to this definition.

The Document may include Warranty Disclaimers next to the notice which states that this License applies to the Document. These Warranty Disclaimers are considered to be included by

reference in this License, but only as regards disclaiming warranties: any other implication that these Warranty Disclaimers may have is void and has no effect on the meaning of this License.

2. VERBATIM COPYING

You may copy and distribute the Document in any medium, either commercially or noncommercially, provided that this License, the copyright notices, and the license notice saying this License applies to the Document are reproduced in all copies, and that you add no other conditions whatsoever to those of this License. You may not use technical measures to obstruct or control the reading or further copying of the copies you make or distribute. However, you may accept compensation in exchange for copies. If you distribute a large enough number of copies you must also follow the conditions in section 3.

You may also lend copies, under the same conditions stated above, and you may publicly display copies.

3. COPYING IN QUANTITY

If you publish printed copies (or copies in media that commonly have printed covers) of the Document, numbering more than 100, and the Document's license notice requires Cover Texts, you must enclose the copies in covers that carry, clearly and legibly, all these Cover Texts: Front-Cover Texts on the front cover, and Back-Cover Texts on the back cover. Both covers must also clearly and legibly identify you as the publisher of these copies. The front cover must present the full title with all words of the title equally prominent and visible. You may add other material on the covers in addition. Copying with changes limited to the covers, as long as they preserve the title of the Document and satisfy these conditions, can be treated as verbatim copying in other respects.

If the required texts for either cover are too voluminous to fit legibly, you should put the first ones listed (as many as fit reasonably) on the actual cover, and continue the rest onto adjacent pages.

If you publish or distribute Opaque copies of the Document numbering more than 100, you must either include a machine-readable Transparent copy along with each Opaque copy, or state in or with each Opaque copy a computer-network location from which the general network-using public has access to download using public-standard network protocols a complete Transparent copy of the Document, free of added material. If you use the latter option, you must take reasonably prudent steps, when you begin distribution of Opaque copies in quantity, to ensure that this Transparent copy will remain thus accessible at the stated location until at least one year after the last time you distribute an Opaque copy (directly or through your agents or retailers) of that edition to the public.

It is requested, but not required, that you contact the authors of the Document well before redistributing any large number of copies, to give them a chance to provide you with an updated version of the Document.

4. MODIFICATIONS

You may copy and distribute a Modified Version of the Document under the conditions of sections 2 and 3 above, provided that you release the Modified Version under precisely this License, with the Modified Version filling the role of the Document, thus licensing distribution and modification of the Modified Version to whoever possesses a copy of it. In addition, you must do these things in the Modified Version:

A. Use in the Title Page (and on the covers, if any) a title distinct from that of the Document, and from those of previous versions (which should, if there were any, be listed in the History section of the Document). You may use the same title as a previous version if the original publisher of that version gives permission.

B List on the Title Page, as authors, one or more persons or entities responsible for authorship of the modifications in the Modified Version, together with at least five of the principal authors of the Document (all of its principal authors, if it has fewer than five), unless they release you from this requirement.

C. State on the Title page the name of the publisher of the Modified Version, as the publisher.

D. Preserve all the copyright notices of the Document.

E. Add an appropriate copyright notice for your modifications adjacent to the other copyright notices.

F. Include, immediately after the copyright notices, a license notice giving the public permission to use the Modified Version under the terms of this License, in the form shown in the Addendum below.

G. Preserve in that license notice the full lists of Invariant Sections and required Cover Texts given in the Document's license notice.

H. Include an unaltered copy of this License.

I. Preserve the section Entitled "History", Preserve its Title, and add to it an item stating at least the title, year, new authors, and publisher of the Modified Version as given on the Title Page. If there is no section Entitled "History" in the Document, create one stating the title, year, authors, and publisher of the Document as given on its Title Page, then add an item describing the Modified Version as stated in the previous sentence.

J. Preserve the network location, if any, given in the Document for public access to a Transparent copy of the Document, and likewise the network locations given in the Document for previous versions it was based on. These may be placed in the "History" section. You may omit a network location for a work that was published at least four years before the Document itself, or if the original publisher of the version it refers to gives permission.

K. For any section Entitled "Acknowledgements" or "Dedications", Preserve the Title of the section, and preserve in the section all the substance and tone of each of the contributor acknowledgements and/or dedications given therein.

L. Preserve all the Invariant Sections of the Document, unaltered in their text and in their titles. Section numbers or the equivalent are not considered part of the section titles.

M. Delete any section Entitled "Endorsements". Such a section may not be included in the Modified Version.

N. Do not retitle any existing section to be Entitled "Endorsements" or to conflict in title with any Invariant Section.

O. Preserve any Warranty Disclaimers.

If the Modified Version includes new front-matter sections or appendices that qualify as Secondary Sections and contain no material copied from the Document, you may at your option designate some or all of these sections as invariant. To do this, add their titles to the list of Invariant Sections in the Modified Version's license notice. These titles must be distinct from any other section titles.

You may add a section Entitled "Endorsements", provided it contains nothing but endorsements of your Modified Version by various parties--for example, statements of peer review or that the text has been approved by an organization as the authoritative definition of a standard.

You may add a passage of up to five words as a Front-Cover Text, and a passage of up to 25 words as a Back-Cover Text, to the end of the list of Cover Texts in the Modified Version. Only one passage of Front-Cover Text and one of Back-Cover Text may be added by (or through arrangements made by) any one entity. If the Document already includes a cover text for the same cover, previously added by you or by arrangement made by the same entity you are acting on behalf of, you may not add another; but you may replace the old one, on explicit permission from the previous publisher that added the old one.

The author(s) and publisher(s) of the Document do not by this License give permission to use their names for publicity for or to assert or imply endorsement of any Modified Version.

5. COMBINING DOCUMENTS

You may combine the Document with other documents released under this License, under the terms defined in section 4 above for modified versions, provided that you include in the combination all of the Invariant Sections of all of the original documents, unmodified, and list them all as Invariant Sections of your combined work in its license notice, and that you preserve all their Warranty Disclaimers.

The combined work need only contain one copy of this License, and multiple identical Invariant Sections may be replaced with a single copy. If there are multiple Invariant Sections with the same name but different contents, make the title of each such section unique by adding at the end of it, in parentheses, the name of the original author or publisher of that section if known, or else a unique number. Make the same adjustment to the section titles in the list of Invariant Sections in the license notice of the combined work.

In the combination, you must combine any sections Entitled "History" in the various original documents, forming one section Entitled "History"; likewise combine any sections Entitled "Acknowledgements", and any sections Entitled "Dedications". You must delete all sections Entitled "Endorsements".

6. COLLECTIONS OF DOCUMENTS

You may make a collection consisting of the Document and other documents released under this License, and replace the individual copies of this License in the various documents with a single copy that is included in the collection, provided that you follow the rules of this License for verbatim copying of each of the documents in all other respects.

You may extract a single document from such a collection, and distribute it individually under this License, provided you insert a copy of this License into the extracted document, and follow this License in all other respects regarding verbatim copying of that document.

7. AGGREGATION WITH INDEPENDENT WORKS

A compilation of the Document or its derivatives with other separate and independent documents or works, in or on a volume of a storage or distribution medium, is called an "aggregate" if the copyright resulting from the compilation is not used to limit the legal rights of the compilation's users beyond what the individual works permit. When the Document is included in an aggregate, this License does not apply to the other works in the aggregate which are not themselves derivative works of the Document.

If the Cover Text requirement of section 3 is applicable to these copies of the Document, then if the Document is less than one half of the entire aggregate, the Document's Cover Texts may be placed on covers that bracket the Document within the aggregate, or the electronic equivalent of covers if the Document is in electronic form. Otherwise they must appear on printed covers that bracket the whole aggregate.

8. TRANSLATION

Translation is considered a kind of modification, so you may distribute translations of the Document under the terms of section 4. Replacing Invariant Sections with translations requires special permission from their copyright holders, but you may include translations of some or all Invariant Sections in addition to the original versions of these Invariant Sections. You may include a translation of this License, and all the license notices in the Document, and any Warranty Disclaimers, provided that you also include the original English version of this License and the original versions of those notices and disclaimers. In case of a disagreement between the translation and the original version of this License or a notice or disclaimer, the original version will prevail.

If a section in the Document is Entitled "Acknowledgements", "Dedications", or "History", the requirement (section 4) to Preserve its Title (section 1) will typically require changing the actual title.

9. TERMINATION

You may not copy, modify, sublicense, or distribute the Document except as expressly provided for under this License. Any other attempt to copy, modify, sublicense or distribute the Document is void, and will automatically terminate your rights under this License. However, parties who have received copies, or rights, from you under this License will not have their licenses terminated so long as such parties remain in full compliance.

10. FUTURE REVISIONS OF THIS LICENSE

The Free Software Foundation may publish new, revised versions of the GNU Free Documentation License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns. See http://www.gnu.org/copyleft/.

Each version of the License is given a distinguishing version number. If the Document specifies that a particular numbered version of this License "or any later version" applies to it, you have the option of following the terms and conditions either of that specified version or of any later version that has been published (not as a draft) by the Free Software Foundation. If the Document does not specify a version number of this License, you may choose any version ever published (not as a draft) by the Free Software Foundation.

ADDENDUM: How to use this License for your documents

To use this License in a document you have written, include a copy of the License in the document and put the following copyright and license notices just after the title page:

If you have Invariant Sections, Front-Cover Texts and Back-Cover Texts, replace the "with...Texts." line with this:

with the Invariant Sections being LIST THEIR TITLES, with the
Front-Cover Texts being LIST, and with the Back-Cover Texts being LIST.

If you have Invariant Sections without Cover Texts, or some other combination of the three, merge those two alternatives to suit the situation.

If your document contains nontrivial examples of program code, we recommend releasing these examples in parallel under your choice of free software license, such as the GNU General Public License, to permit their use in free software.

Index

129

```
-----------------------------------------------------------------------------
-- File name:        BussCntC.vhd
--
-- Description:      Address & data bus control logic.
--
-- Created on:       Thu Feb 14 10:03:06 2002
--
-- Source version:   VMEcore(tm) VER: 1.0.0
--
-- Interface:        Type:          Attribute(s):
--                   -----          -------------
--                   LOCAL:   LASIZE:19 LDSIZE:32
--                   SLAVE:   A24:SGL D32 D16(EO) D08(O)
--                   SLAVE:   BERR* SIGNAL SUPPORTED
--
-- Copyright:        Copyright (C) 2002 SILICORE CORPORATION.  This
--                   document is protected by U.S. and international
--                   copyright laws.  No part of this document may be
--                   reproduced by any means without the express,
--                   prior, written consent of SILICORE CORPORATION
--                   (Corcoran, MN  USA).  For more information
--                   please refer to the license agreement.
--
-----------------------------------------------------------------------------
-----------------------------------------------------------------------------
-- Load the IEEE 1164 library and make it visible.
-----------------------------------------------------------------------------
library ieee;
use ieee.std_logic_1164.all;

-----------------------------------------------------------------------------
-- Entity / port definition.
-----------------------------------------------------------------------------
entity BUSSCNTC is

    port(
        A:        out std_logic_vector( 1 downto 1 );
        ALOAD:    out std_logic;
        D16EO:    out std_logic;
        DLOAD:    out std_logic;
        SLAD:     in  std_logic_vector( 1 downto 1 );
        SLALD:    in  std_logic;
        SLD16EO:  in  std_logic;
        SLDLD:    in  std_logic;
        SLVLD:    in  std_logic;
```

```vhdl
    VLOAD:          out std_logic
    );
end entity BUSSCNTC;

architecture BUSSCNTC1 of BUSSCNTC is

----------------------------------------------------------------
-- Architecture definition.
----------------------------------------------------------------
begin

    ----------------------------------------------------------------
    -- Bus control logic.
    ----------------------------------------------------------------

    BUS_CONTROL: process( SLAD, SLALD, SLD16EO, SLDLD, SLVLD )
    begin

        ALOAD   <= SLALD;

        DLOAD   <= SLDLD;

        VLOAD   <= SLVLD;

        D16EO   <= SLD16EO;

        A(1)    <= SLAD(1);

    end process BUS_CONTROL;

end architecture BUSSCNTC1;
```

```vhdl
-- ------------------------------------------------------------------
-- File name:        LoclAdrC.vhd
-- |
-- Description:      Local address bus generator.
-- |
-- Created on:       Thu Feb 14 10:03:06 2002
-- |
-- Source version:   VMEcore(tm) VER: 1.0.0
-- |
-- Interface:        Type:           Attribute(s):
-- |                 ------          -------------
-- |                 LOCAL:          LASIZE:19 LDSIZE:32
-- |                 SLAVE:          A24:SGL D32 D16(EO) D08(O)
-- |                 SLAVE:          BERR* SIGNAL SUPPORTED
-- |
-- Copyright:        Copyright (C) 2002 SILICORE CORPORATION.  This
-- |                 document is protected by U.S. and international
-- |                 copyright laws.  No part of this document may be
-- |                 reproduced by any means without the express,
-- |                 prior, written consent of SILICORE CORPORATION
-- |                 (Corcoran, MN  USA).  For more information
-- |                 please refer to the license agreement.
-- |
-- ------------------------------------------------------------------

-- ------------------------------------------------------------------
-- Load the IEEE 1164 library and make it visible.
-- ------------------------------------------------------------------

library ieee;
use ieee.std_logic_1164.all;

-- ------------------------------------------------------------------
-- Entity / port definition.
-- ------------------------------------------------------------------

entity LOCLADRC is

   port(
      ADR_O:      out std_logic_vector( 18 downto  2 );
      ALOAD:      in  std_logic;
      CLK_I:      in  std_logic;
      VA_I:       in  std_logic_vector( 18 downto  2 )
      );

end entity LOCLADRC;
```

```
-------------------------------------------------------------------
-- Architecture definition.
-------------------------------------------------------------------
-------------------------------------------------------------------

architecture LOC1ADRC1 of LOC1ADRC is

begin

-------------------------------------------------------------------
--  Process for local address registers and/or counters.
-------------------------------------------------------------------

ADR_REG: process ( CLK_I )
begin
  if( rising_edge(CLK_I) ) then if( ALOAD = '1' ) then ADR_O( 2) <= VA_I( 2); end if; end if;
  if( rising_edge(CLK_I) ) then if( ALOAD = '1' ) then ADR_O( 3) <= VA_I( 3); end if; end if;
  if( rising_edge(CLK_I) ) then if( ALOAD = '1' ) then ADR_O( 4) <= VA_I( 4); end if; end if;
  if( rising_edge(CLK_I) ) then if( ALOAD = '1' ) then ADR_O( 5) <= VA_I( 5); end if; end if;
  if( rising_edge(CLK_I) ) then if( ALOAD = '1' ) then ADR_O( 6) <= VA_I( 6); end if; end if;
  if( rising_edge(CLK_I) ) then if( ALOAD = '1' ) then ADR_O( 7) <= VA_I( 7); end if; end if;
  if( rising_edge(CLK_I) ) then if( ALOAD = '1' ) then ADR_O( 8) <= VA_I( 8); end if; end if;
  if( rising_edge(CLK_I) ) then if( ALOAD = '1' ) then ADR_O( 9) <= VA_I( 9); end if; end if;
  if( rising_edge(CLK_I) ) then if( ALOAD = '1' ) then ADR_O(10) <= VA_I(10); end if; end if;
  if( rising_edge(CLK_I) ) then if( ALOAD = '1' ) then ADR_O(11) <= VA_I(11); end if; end if;
  if( rising_edge(CLK_I) ) then if( ALOAD = '1' ) then ADR_O(12) <= VA_I(12); end if; end if;
  if( rising_edge(CLK_I) ) then if( ALOAD = '1' ) then ADR_O(13) <= VA_I(13); end if; end if;
  if( rising_edge(CLK_I) ) then if( ALOAD = '1' ) then ADR_O(14) <= VA_I(14); end if; end if;
  if( rising_edge(CLK_I) ) then if( ALOAD = '1' ) then ADR_O(15) <= VA_I(15); end if; end if;
  if( rising_edge(CLK_I) ) then if( ALOAD = '1' ) then ADR_O(16) <= VA_I(16); end if; end if;
  if( rising_edge(CLK_I) ) then if( ALOAD = '1' ) then ADR_O(17) <= VA_I(17); end if; end if;
  if( rising_edge(CLK_I) ) then if( ALOAD = '1' ) then ADR_O(18) <= VA_I(18); end if; end if;

end process ADR_REG;

end architecture LOC1ADRC1;
```

```
-- File name:        LoclDatC.vhd
--
-- Description:      Data multiplexor and/or register for DAT_O().
--
-- Created on:       Thu Feb 14 10:03:06 2002
--
-- Source version:   VMEcore(tm) VER: 1.0.0
--
-- Interface:        Type:          Attribute(s):
--                   -----          -------------
--                   LOCAL:         LASIZE:19 LDSIZE:32
--                   SLAVE:         A24:SGL D32 D16(EO) D08(O)
--                   SLAVE:         BERR* SIGNAL SUPPORTED
--
-- Copyright:        Copyright (C) 2002 SILICORE CORPORATION.  This
--                   document is protected by U.S. and international
--                   copyright laws.  No part of this document may be
--                   reproduced by any means without the express,
--                   prior, written consent of SILICORE CORPORATION
--                   (Corcoran, MN  USA).  For more information
--                   please refer to the license agreement.
--

-- Load the IEEE 1164 library and make it visible.

library ieee;
use ieee.std_logic_1164.all;


-- Entity / port definition.

entity LOCLDATC is

   port(
        CLK_I:       in  std_logic;
        D16EO:       in  std_logic;
        DAT_O:       out std_logic_vector( 31 downto  0 );
        DLOAD:       in  std_logic;
        VD_I:        in  std_logic_vector( 31 downto  0 )
        );

end entity LOCLDATC;
```

```vhdl
-------------------------------------------------------
-- Architecture definition.
-------------------------------------------------------
-------------------------------------------------------
architecture LOCLDATC1 of LOCLDATC is

-------------------------------------------------------
-- Signal definition(s).
-------------------------------------------------------

    signal   EN1:            std_logic;

begin

-------------------------------------------------------
-- WORD1: DAT_O( 31 downto 16 ).
-------------------------------------------------------
WORD1: process ( CLK_I, D16EO, EN1 )
begin

    EN1 <= D16EO;

    if ( rising_edge(CLK_I) ) then
        if ( DLOAD = '1' ) then

            case EN1 is
                when        '0'  => DAT_O( 31 downto 16 )  <= VD_I( 31 downto 16 )  );
                when others => DAT_O( 31 downto 16 )  <= VD_I( 15 downto   0 )  );
            end case;

        end if;
    end if;

end process WORD1;

-------------------------------------------------------
-- WORD0: DAT_O( 15 downto   0 ).
-------------------------------------------------------
WORD0: process ( CLK_I )
begin

    if ( rising_edge(CLK_I) ) then
        if ( DLOAD = '1' ) then
            DAT_O( 15 downto   0 )  <= VD_I( 15 downto   0 )  ;
        end if;
    end if;
```

```vhdl
        end if;

    end process WORD0;

end architecture LOCLDATC1;
```

```
-- ---------------------------------------------------------------------------
--
-- File name:       SlavCntC.vhd
--
-- Description:     SLAVE SlavCnt logic.
--
-- Created on:      Thu Feb 14 10:03:06 2002
--
-- Source version:  VMEcore(tm) VER: 1.0.0
--
-- Interface:
--
--              Type:                       Attribute(s):
--              ------                      -------------
--              LOCAL:  LASIZE:19 LDSIZE:32
--              SLAVE:  A24:SGL D32 D16(EO) D08(O)
--              SLAVE:  BERR* SIGNAL SUPPORTED
--
-- Copyright:   Copyright (C) 2002 SILICORE CORPORATION.  This
--              document is protected by U.S. and international
--              copyright laws.  No part of this document may be
--              reproduced by any means without the express,
--              prior, written consent of SILICORE CORPORATION
--              (Corcoran, MN  USA).  For more information
--              please refer to the license agreement.
--
--
-- ---------------------------------------------------------------------------

-- Load the IEEE 1164 library and make it visible.

library ieee;
use ieee.std_logic_1164.all;

-- Entity / port definition.

entity SLAVCNTC is

    port(
        ACK_I:      in  std_logic;
        ADC:        in  std_logic;
        CLK_I:      in  std_logic;
        CYC_O:      out std_logic;
        ERR_I:      in  std_logic;
        RST_I:      in  std_logic;
        SEL_O:      out std_logic_vector( 3 downto 0 );
        SLAD:       out std_logic_vector( 1 downto 1 );
        SLALD:      out std_logic;
```

```vhdl
        SLD16EO:    out std_logic;
        SLDLD:      out std_logic;
        SLVLD:      out std_logic;
        STB_O:      out std_logic;
        VA_I:       in  std_logic_vector( 1 downto 1 );
        VNAS_I:     in  std_logic;
        VNBERR_O:   out std_logic;
        VNDS0_I:    in  std_logic;
        VNDS1_I:    in  std_logic;
        VNDTACK_O:  out std_logic;
        VNLWORD_I:  in  std_logic;
        VNWRITE_I:  in  std_logic;
        WE_O:       out std_logic
    );

end entity SLAVCNTC;


---------------------------------------
-- Architecture definition.
---------------------------------------

architecture SLAVCNTC1 of SLAVCNTC is

---------------------------------------
-- Signal definition(s).
---------------------------------------

    signal  A:          std_logic_vector( 1 downto 1 );
    signal  ACK:        std_logic;
    signal  ALOAD:      std_logic;
    signal  AQ:         std_logic_vector( 1 downto 1 );
    signal  AS:         std_logic;
    signal  BERR:       std_logic;
    signal  BMC:        std_logic;
    signal  CYC:        std_logic;
    signal  D16EO:      std_logic;
    signal  D32:        std_logic;
    signal  DLOAD:      std_logic;
    signal  DREAD:      std_logic;
    signal  DS:         std_logic;
    signal  DTACK:      std_logic;
    signal  DTKCD:      std_logic;
    signal  IHLD:       std_logic;
    signal  MUX:        std_logic;
    signal  NAS:        std_logic;
    signal  NDS0:       std_logic;
    signal  NDS1:       std_logic;
    signal  NLWORD:     std_logic;
    signal  P:          std_logic_vector( 2 downto 0 );
```

```vhdl
signal  SEL:     std_logic_vector( 3 downto 0 );
signal  SELD:    std_logic;
signal  STB:     std_logic;

begin

-------------------------------------------------------
-- Common logic used by other processes.
-------------------------------------------------------

TERMINATOR: process ( ACK_I, ERR_I )
begin

    ACK  <= ACK_I or ERR_I;

end process TERMINATOR;

-------------------------------------------------------
-- Synchronizer for address and data strobes.
-------------------------------------------------------

SYNC: process ( CLK_I, NAS, NDS0, NDS1 )
begin

    if ( rising_edge(CLK_I) )  then NAS  <= VNAS_I;   end if;
    if ( rising_edge(CLK_I) )  then NDS0 <= VNDS0_I;  end if;
    if ( rising_edge(CLK_I) )  then NDS1 <= VNDS1_I;  end if;

    AS  <= not(NAS);
    DS  <= not(NDS0) or not(NDS1);

end process SYNC;

-------------------------------------------------------
-- Sequencer.
-------------------------------------------------------

SEQU: process ( ALOAD, CLK_I, DLOAD, DS, MUX, P )
begin

-------------------------------------------------------
-- Internal signals.
-------------------------------------------------------

    IHLD <= '0';
    BMC  <= '0';
    MUX  <= '0';
```

```
--------------------
-- State machine.
--------------------

if( rising_edge(CLK_I) ) then

  P(0) <= ( not(P(2))                                                      and            AS  and  DS  )
     or ( P(2)     and not(P(1))              and        P(0)  and IHLD    and            AS  and  DS  );

  P(1) <= ( not(P(2))   and P(1)                                           and            AS  and  DS  )
     or ( not(P(2))   and P(1)                                             and            AS  and not(DS))
     or ( not(P(2))   and not(P(1))                                        and            AS  and not(DS))
     or (               not(P(1))            and not(P(0))                 and  not(AS)       and  DS  )
     or (                          P(1)                                    and            AS  and  DS  )
     or ( P(2)     and not(P(1))              and not(P(0))                 and  not(AS)       and  DS  )
     or ( P(2)     and P(1)                   and        P(0)  and IHLD    and            AS  and  DS  )
     or ( not(P(2))   and not(P(1))          and not(P(0)) and IHLD        and            AS  and  DS  );

  P(2) <= ( P(2)     and not(P(1))                                         and            AS  and  DS  )
     or ( not(P(2))   and P(1)                                             and  not(AS)       and  DS  )
     or ( not(P(2))   and P(1)                                             and  not(AS)       and  DS  )
     or ( P(2)                                and        P(0)              and            AS  and  DS  )
     or (                          P(1)      and not(P(0))                 and            AS  and  DS  )
     or (                          P(1)      and not(P(0))                 and  not(AS)       and  DS  )
     or ( not(P(2))   and P(1)                and        P(0)              and            AS  and not(DS))
     or ( P(2)     and not(P(1))                                          and            AS  )
     or ( P(2)     and P(1)                   and        P(0)  and IHLD    and            AS  )
     or ( not(P(2))   and not(P(1))          and not(P(0)) and IHLD        and            AS  and  DS  );

end if;

--------------------
-- Control signals.
--------------------

ALOAD <= ( not(P(2)) and not(P(1)) and not(P(0))              and   DS   and not(IHLD) );

SLALD <= ALOAD;

DLOAD <= ( not(P(2)) and not(P(1)) and not(P(0)) and not(MUX) and   DS   and not(IHLD) )
    or ( not(P(2)) and P(1)  and not(P(0))                    and   DS   and not(IHLD) )
    or ( P(2)  and not(P(1)) and not(P(0))                    and   DS   and not(IHLD) );

SLDLD <= DLOAD;

end process SEQU;
```

```vhdl
-----------------------------------------------------------------------------------
-- Data cycle function generator.
-----------------------------------------------------------------------------------

DCYCL: process ( A, D16EO, NLWORD, VNDS0_I, VNDS1_I )
begin

D32       <= not(VNDS1_I) and not(VNDS0_I) and not(NLWORD) and not(A(1)) ;

D16EO     <= ( not(VNDS1_I)  and NLWORD )
          or ( not(VNDS0_I)  and NLWORD ) ;

SLD16EO   <= D16EO;

end process DCYCL;

-----------------------------------------------------------------------------------
-- Select signal(s).
-----------------------------------------------------------------------------------

SEL_GENR: process ( CLK_I, SEL )
begin

if ( rising_edge(CLK_I) ) then

SEL(3)  <= ( DLOAD and ADC and D32 )
        or ( DLOAD and ADC and D16EO and not(VNDS1_I) and not(A(1)) )
        or ( SEL(3) and DS  and not(ACK) and not(RST_I) ) ;

SEL_O(3) <= SEL(3) ;

end if;

if ( rising_edge(CLK_I) ) then

SEL(2)  <= ( DLOAD and ADC and D32 )
        or ( DLOAD and ADC and D16EO and not(VNDS0_I) and not(A(1)) )
        or ( SEL(2) and DS  and not(ACK) and not(RST_I) ) ;

SEL_O(2) <= SEL(2) ;

end if;

if ( rising_edge(CLK_I) ) then

SEL(1)  <= ( DLOAD and ADC and D32 )
```

```vhdl
                    or ( DLOAD and ADC and D16EO and not(VNDS1_I) and A(1) )
                    or ( SEL(1) and DS and not(ACK) and not(RST_I) );

        end if;

        SEL_O(1) <= SEL(1);

        if( rising_edge(CLK_I) ) then

            SEL(0) <= ( DLOAD and ADC and D32 )
                    or ( DLOAD and ADC and D16EO and not(VNDS0_I) and A(1) )
                    or ( SEL(0) and DS and not(ACK) and not(RST_I) );

        end if;

        SEL_O(0) <= SEL(0);

    end process SEL_GENR;

    --------------------------------------------------------------------
    -- Strobe(s).
    --------------------------------------------------------------------

    STROBES: process( ACK, AS, CLK_I, CYC, DS, STB )
    begin

        --------------------------------------------------------------------
        -- CYC_O.
        --------------------------------------------------------------------

        if( rising_edge(CLK_I) ) then

            CYC <= (ALOAD and (ADC)) or (CYC and AS and not(RST_I));

        end if;

        CYC_O <= CYC and AS;

        --------------------------------------------------------------------
        -- STB_O.
        --------------------------------------------------------------------

        if( rising_edge(CLK_I) ) then

            STB <= (DLOAD and (ADC)) or (STB and DS and not(ACK) and not(RST_I));
```

```vhdl
    end if;

    STB_O <= STB and DS;

    --------------------------------------------------------
    -- WE_O.
    --------------------------------------------------------
    if( rising_edge(CLK_I) ) then if(DLOAD = '1') then

        WE_O <= not(VNWRITE_I) ;

    end if; end if;

end process STROBES;

--------------------------------------------------------
-- DTACK generator.
--------------------------------------------------------
DTACK_GENR: process( ACK_I, ADC, CLK_I, DS, DTACK, DTKCD, STB, VNWRITE_I )
begin

    SELD  <= ADC and DS;

    DTKCD <= VNWRITE_I;

    if( rising_edge(CLK_I) ) then

        DREAD <= ( not(RST_I) and not(DTACK) and not(DREAD) and ACK_I and DTKCD and DS and STB and SELD );

        DTACK <= ( not(RST_I) and not(DTACK) and not(DREAD) and ACK_I and not(DTKCD) and DS and SELD )
            or ( not(RST_I) and not(DTACK) and     DREAD                      and DS and SELD )
            or ( not(RST_I) and     DTACK   and not(DREAD)                     and DS and SELD );

        BERR  <= ( not(RST_I) and not(BERR) and ERR_I and DS )
            or ( not(RST_I) and     BERR              and DS );

    end if;

    VNDTACK_O <= not(DTACK and DS);
    VNBERR_O  <= not(BERR and DS);

    SLVLD <= STB and ACK_I;

end process DTACK_GENR;
```

```
---------------------------------------------
-- Miscellaneous address control signals.
---------------------------------------------

misc_addr: process( A, ALOAD, AQ, VA_I, VNLWORD_I )
begin

    if ALOAD = '1' then NLWORD <= VNLWORD_I; end if;

    if ALOAD = '1' then AQ(1) <= VA_I(1); end if;

    A(1) <= AQ(1);

    SLAD(1) <= A(1);

end process misc_addr;


end architecture SLAVCNTC1;
```

```
-- File name:        SlavDecC.vhd
--
-- Description:      Slave address decode logic.
--
-- Created on:       Thu Feb 14 10:03:06 2002
--
-- Source version:   VMEcore(tm) VER: 1.0.0
--
-- Interface:
--
--    Type:                             Attribute(s):
--    ----                              -------------
--    LOCAL:    LASIZE:19 LDSIZE:32
--    SLAVE:    A24:SGL D32 D16(EO) D08(O)
--    SLAVE:    BERR* SIGNAL SUPPORTED
--
--
-- Copyright:        Copyright (C) 2002 SILICORE CORPORATION.  This
--                   document is protected by U.S. and international
--                   copyright laws.  No part of this document may be
--                   reproduced by any means without the express,
--                   prior, written consent of SILICORE CORPORATION
--                   (Corcoran, MN  USA).  For more information
--                   please refer to the license agreement.
--
```

-- Load the IEEE 1164 library and make it visible.

```
library ieee;
use ieee.std_logic_1164.all;

-- Entity / port definition.

entity SLAVDECC is

   port (

      A24SGL_O:    out std_logic;
      ADC:         out std_logic;
      ALOAD:       in  std_logic;
      CLK_I:       in  std_logic;
      SAC24_I:     in  std_logic_vector( 23 downto 19 );
      SAE24_I:     in  std_logic;
      VAM_I:       in  std_logic_vector(  5 downto  0 );
      VA_I:        in  std_logic_vector( 23 downto 19 );
      VNIACK_I:    in  std_logic
```

```vhdl
                               );

    end entity SLAVDECC;

    ---------------------------------------------------------------
    -- Architecture definition.
    ---------------------------------------------------------------

    architecture SLAVDECC1 of SLAVDECC is

    ---------------------------------------------------------------
    -- Signal definition(s).
    ---------------------------------------------------------------

    signal   A24SGL:          std_logic;
    signal   AD:              std_logic;
    signal   SCMP24:          std_logic;


begin

    ---------------------------------------------------------------
    -- Address modifier code, decode logic.
    ---------------------------------------------------------------

    AM_CODES: process( CLK_I, VAM_I, VNIACK_I )
    begin

        A24SGL <= ( VNIACK_I and VAM_I(5) and VAM_I(4) and VAM_I(3) and VAM_I(2) and VAM_I(1)
    and not(VAM_I(0)) )
              or ( VNIACK_I and VAM_I(5) and VAM_I(4) and VAM_I(3) and VAM_I(2) and not(VAM_I(1))
    and     VAM_I(0) )
              or ( VNIACK_I and VAM_I(5) and VAM_I(4) and VAM_I(3) and not(VAM_I(2)) and VAM_I(1)
    and not(VAM_I(0)) )
              or ( VNIACK_I and VAM_I(5) and VAM_I(4) and VAM_I(3) and not(VAM_I(2)) and not(VAM_I(1))
    and     VAM_I(0) );

        if( rising_edge(CLK_I) ) then if( ALOAD = '1' ) then A24SGL_O <= A24SGL; end if; end if;

    end process AM_CODES;

    ---------------------------------------------------------------
    -- SLAVE A24 address comparator.
    ---------------------------------------------------------------

    SLAVE_A24CMP: process( SAC24_I, SAE24_I, VA_I )
    begin
```

```vhdl
    SCMP24 <=    ( SAC24_I(23)   xnor  VA_I(23) )
        and  ( SAC24_I(22)   xnor  VA_I(22) )
        and  ( SAC24_I(21)   xnor  VA_I(21) )
        and  ( SAC24_I(20)   xnor  VA_I(20) )
        and  ( SAC24_I(19)   xnor  VA_I(19) )
        and    SAE24_I ;

end process SLAVE_A24CMP;

--------------------------------------------
-- ADC decoder.
--------------------------------------------

ADC_DEC: process ( A24SGL, AD,  SCMP24 )
begin

    AD   <=  ( (A24SGL)  and  SCMP24 ) ;

    ADC  <= AD;

end process ADC_DEC;

end architecture SLAVDECC1;
```

VMEcore(tm) Interface Writer, Version: 1.0.0
Run time: Thu Feb 14 10:03:06 2002
Option file attribute check:
Option file attributes are okay.

```
;
; Generic test file
;
;
;MASTER:   A24:BLT A32:BLT A40:BLT A64:BLT MD32 D32 D16(EO)
;MASTER:   A24:CSR D32:UAT D32 D16(EO)
;MASTER:   A16:LCK A24:LCK A32:LCK A40:LCK A64:LCK
;MASTER:   A24:MBT A32:MBT A64:MBT
;MASTER:   A16:SGL A24:SGL A32:SGL A40:SGL A64:SGL MD32 D32:UAT D32 D16(EO)
;MASTER:   A64:SIZE:64 A40:SIZE:40 A32:SIZE:32 A24:SIZE:24 A16:SIZE:16
;SLAVE:    A24:BLT A32:BLT A40:BLT A64:BLT MD32 D32 D16(EO)
;SLAVE:    A24:CSR D32:UAT D32 D16(EO) D08(O)
;SLAVE:    A16:LCK A24:LCK A32:LCK A40:LCK A64:LCK
;SLAVE:    A24:MBT A32:MBT A64:MBT
;SLAVE:    A16:SGL A24:SGL A32:SGL A40:SGL A64:SGL MD32 D32:UAT D32 D16(EO) D08(O)
;SLAVE:    BERR RETRY
;LOCMON:   A24:BLT A32:BLT A40:BLT A64:BLT
;LOCMON:   A24:CSR
;LOCMON:   A16:LCK A24:LCK A32:LCK A40:LCK A64:LCK
;LOCMON:   A24:MBT A32:MBT A64:MBT
;LOCMON:   A16:SGL A24:SGL A32:SGL A40:SGL A64:SGL
;IHAND:    D32 D16 D08(O)
;IREQ:     D32 D16 D08(O)
;LOCAL:    LASIZE:20 LDSIZE:64

SLAVE:    A24:SGL
SLAVE:    D32 D16(EO) D08(O)
SLAVE:    BERR
LOCAL:    LASIZE:19 LDSIZE:32

FREQ:  33.3
```

```vhdl
---------------------------------------------------------------------
-- File name:        VmebADbC.vhd
--
-- Description:      Data multiplexor and/or register for VD_O().
--
-- Created on:       Thu Feb 14 10:03:06 2002
--
-- Source version:   VMEcore(tm) VER: 1.0.0
--
-- Interface:        Type:            Attribute(s):
--                   -----            -------------
--                   LOCAL:    LASIZE:19 LDSIZE:32
--                   SLAVE:    A24:SGL D32 D16(EO) D08(O)
--                   SLAVE:    BERR* SIGNAL SUPPORTED
--
---------------------------------------------------------------------
-- Copyright:        Copyright (C) 2002 SILICORE CORPORATION.  This
--                   document is protected by U.S. and international
--                   copyright laws.  No part of this document may be
--                   reproduced by any means without the express,
--                   prior, written consent of SILICORE CORPORATION
--                   (Corcoran, MN  USA).  For more information
--                   please refer to the license agreement.
---------------------------------------------------------------------
---------------------------------------------------------------------
-- Load the IEEE 1164 library and make it visible.
---------------------------------------------------------------------
library ieee;
use ieee.std_logic_1164.all;

---------------------------------------------------------------------
-- Entity / port definition.
---------------------------------------------------------------------
entity VMEBADBC is

    port(

        A:       in  std_logic_vector(  1 downto  1 );
        CLK_I:   in  std_logic;
        D16EO:   in  std_logic;
        DAT_I:   in  std_logic_vector( 31 downto  0 );
        VD_O:    out std_logic_vector( 31 downto  0 );
        VLOAD:   in  std_logic

        );

end entity VMEBADBC;
```

```vhdl
-------------------------------------------------------------
-- Architecture definition.
-------------------------------------------------------------
-------------------------------------------------------------

architecture VMEBADBC1 of VMEBADBC is

    -------------------------------------------------------------
    -- Signal definition(s).
    -------------------------------------------------------------
    -------------------------------------------------------------

    signal  DLO:            std_logic;
    signal  DME:            std_logic;

begin

    -------------------------------------------------------------
    -- VD_O( 31 downto 16 ).
    -------------------------------------------------------------
    -------------------------------------------------------------

    VDATO32: process ( CLK_I )
    begin

        if ( rising_edge(CLK_I) ) then
            if ( VLOAD = '1' ) then
                VD_O( 31 downto 16 ) <= DAT_I( 31 downto 16 );
            end if;
        end if;

    end process VDATO32;

    -------------------------------------------------------------
    -- VD_O( 15 downto  8 ).
    -------------------------------------------------------------
    -------------------------------------------------------------

    VDATO16: process ( A, CLK_I, D16EO, DME )
    begin

        DME <= (D16EO and not(A(1)));

        if ( rising_edge(CLK_I) ) then
            if ( VLOAD = '1' ) then

                case DME is
                    when      '0'   => VD_O( 15 downto  8 ) <= DAT_I( 15 downto  8 );
                    when others   => VD_O( 15 downto  8 ) <= DAT_I( 31 downto 24 );
                end case;
```

```vhdl
        end if;
      end if;

  end process VDATO16;

  ---------------------------------
  -- VD_O(  7 downto  0 ).
  ---------------------------------

  VDATO08: process( A, CLK_I, D16EO, DLO )
  begin

    DLO <= (D16EO and not(A(1)));

    if( rising_edge(CLK_I) ) then
      if( VLOAD = '1' ) then

        case DLO is
          when  '0'    => VD_O(  7 downto  0 ) <= DAT_I(  7 downto  0 );
          when others  => VD_O(  7 downto  0 ) <= DAT_I( 23 downto 16 );
        end case;

      end if;
    end if;

  end process VDATO08;

end architecture VMEBADBC1;
```

```vhdl
--  ------------------------------------------------------------------
--
--  File name:          VmeCoreC.vhd
--
--  Description:        VMEcore high-level entity/architecture.
--
--  Created on:         Thu Feb 14 10:03:06 2002
--
--  Source version:     VMEcore(tm) VER: 1.0.0
--
--  ------------------------------------------------------------------
--
--  Interface:          Type:                Attribute(s):
--                      -----                -------------
--                      LOCAL:  LASIZE:19 LDSIZE:32
--                      SLAVE:  A24:SGL D32 D16(EO) D08(O)
--                      SLAVE:  BERR* SIGNAL SUPPORTED
--
--  ------------------------------------------------------------------
--
--  Copyright:          Copyright (C) 2002 SILICORE CORPORATION.  This
--                      document is protected by U.S. and international
--                      copyright laws.  No part of this document may be
--                      reproduced by any means without the express,
--                      prior, written consent of SILICORE CORPORATION
--                      (Corcoran, MN  USA).  For more information
--                      please refer to the license agreement.
--
--  ------------------------------------------------------------------

library ieee;
use ieee.std_logic_1164.all;

-- Load the IEEE 1164 library and make it visible.
--  ------------------------------------------------------------------

-- Entity / port definition.

entity VMECOREC is

    port (
        A24SGL_O:   out std_logic;
        ACK_I:      in  std_logic;
        ADR_O:      out std_logic_vector( 18 downto  2 );
        CLK_I:      in  std_logic;
        CYC_O:      out std_logic;
        DAT_I:      in  std_logic_vector( 31 downto  0 );
        DAT_O:      out std_logic_vector( 31 downto  0 );
        ERR_I:      in  std_logic;
        RST_I:      in  std_logic;
```

```vhdl
        SAC24_I:        in  std_logic_vector( 23 downto 19 );
        SAE24_I:        in  std_logic;
        SEL_O:          out std_logic_vector(  3 downto  0 );
        STB_O:          out std_logic;
        VAM_I:          in  std_logic_vector(  5 downto  0 );
        VA_I:           in  std_logic_vector( 23 downto  1 );
        VD_I:           in  std_logic_vector( 31 downto  0 );
        VD_O:           out std_logic_vector( 31 downto  0 );
        VNAS_I:         in  std_logic;
        VNBERR_O:       out std_logic;
        VNDS0_I:        in  std_logic;
        VNDS1_I:        in  std_logic;
        VNDTACK_O:      out std_logic;
        VNIACK_I:       in  std_logic;
        VNLWORD_I:      in  std_logic;
        VNWRITE_I:      in  std_logic;
        WE_O:           out std_logic
    );

end entity VMECOREC;

-------------------------------------------------------
-- Architecture definition.
-------------------------------------------------------

architecture VMECOREC1 of VMECOREC is

-------------------------------------------------------
-- Component definition(s).
-------------------------------------------------------

component BUSSCNTC
port(
    A:              out std_logic_vector(  1 downto  1 );
    ALOAD:          out std_logic;
    D16EO:          out std_logic;
    DLOAD:          out std_logic;
    SLAD:           in  std_logic_vector(  1 downto  1 );
    SLALD:          in  std_logic;
    SLD16EO:        in  std_logic;
    SLDLD:          in  std_logic;
    SLVLD:          in  std_logic;
    VLOAD:          out std_logic
    );
end component BUSSCNTC;

component LOCLADRC
port(
```

```vhdl
    ADR_O:      out std_logic_vector( 18 downto  2 );
    ALOAD:      in  std_logic;
    CLK_I:      in  std_logic;
    VA_I:       in  std_logic_vector( 18 downto  2 )
  );
end component LOCLADRC;

component LOCLDATC
port(
    CLK_I:      in  std_logic;
    D16EO:      in  std_logic;
    DAT_O:      out std_logic_vector( 31 downto  0 );
    DLOAD:      in  std_logic;
    VD_I:       in  std_logic_vector( 31 downto  0 )
  );
end component LOCLDATC;

component SLAVCNTC
port(
    ACK_I:      in  std_logic;
    ADC:        in  std_logic;
    CLK_I:      in  std_logic;
    CYC_O:      out std_logic;
    ERR_I:      in  std_logic;
    RST_I:      in  std_logic;
    SEL_O:      out std_logic_vector( 3 downto 0 );
    SLAD:       in  std_logic;
    SLAID:      out std_logic_vector( 1 downto 1 );
    SLD16EO:    out std_logic;
    SLDLD:      out std_logic;
    SLVLD:      out std_logic;
    STB_O:      out std_logic;
    VA_I:       in  std_logic_vector( 1 downto 1 );
    VNAS_I:     in  std_logic;
    VNBERR_O:   out std_logic;
    VNDS0_I:    in  std_logic;
    VNDS1_I:    in  std_logic;
    VNDTACK_O:  out std_logic;
    VNLWORD_I:  in  std_logic;
    VNWRITE_I:  in  std_logic;
    WE_O:       out std_logic
  );
end component SLAVCNTC;

component SLAVDECC
port(
    A24SGL_O:   out std_logic;
    ADC:        out std_logic;
    ALOAD:      in  std_logic;
    CLK_I:      in  std_logic;
```

```vhdl
        SAC24_I:        in  std_logic_vector( 23 downto 19 );
        SAE24_I:        in  std_logic;
        VAM_I:          in  std_logic_vector(  5 downto  0 );
        VA_I:           in  std_logic_vector( 23 downto 19 );
        VNIACK_I:       in  std_logic
        );
end component SLAVDECC;

component VMEBADBC
port(
        A:              in  std_logic_vector(  1 downto  1 );
        CLK_I:          in  std_logic;
        D16EO:          in  std_logic;
        DAT_I:          in  std_logic_vector( 31 downto  0 );
        VD_O:           out std_logic_vector( 31 downto  0 );
        VLOAD:          in  std_logic
        );
end component VMEBADBC;

--------------------------------------------------
-- Signal definition(s).
--------------------------------------------------

signal A:               std_logic_vector(  1 downto  1 );
signal ADC:             std_logic;
signal ALOAD:           std_logic;
signal D16EO:           std_logic;
signal DLOAD:           std_logic;
signal SLAD:            std_logic_vector(  1 downto  1 );
signal SLALD:           std_logic;
signal SLD16EO:         std_logic;
signal SLDLD:           std_logic;
signal SLVLD:           std_logic;
signal VLOAD:           std_logic;

--------------------------------------------------

begin

--------------------------------------------------
-- Component map.
--------------------------------------------------

U1: component BUSSCNTC
port map(
        A               => A(  1 downto  1 ),
        ALOAD           => ALOAD,
        D16EO           => D16EO,
        DLOAD           => DLOAD,
        SLAD            => SLAD(  1 downto  1 ),
```

```vhdl
U2: component LOCLADRC
port map(
    ADR_O       => ADR_O( 18 downto  2 ),
    ALOAD       => ALOAD,
    CLK_I       => CLK_I,
    VA_I        => VA_I( 18 downto  2 )
);

U3: component LOCLDATC
port map(
    CLK_I       => CLK_I,
    D16EO       => D16EO,
    DAT_O       => DAT_O( 31 downto  0 ),
    DLOAD       => DLOAD,
    VD_I        => VD_I( 31 downto  0 )
);

U4: component SLAVCNTC
port map(
    ACK_I       => ACK_I,
    ADC         => ADC,
    CLK_I       => CLK_I,
    CYC_O       => CYC_O,
    ERR_I       => ERR_I,
    RST_I       => RST_I,
    SEL_O       => SEL_O( 3 downto  0 ),
    SIAD        => SIAD( 1 downto  1 ),
    SIALD       => SIALD,
    SLD16EO     => SLD16EO,
    SLDLD       => SLDLD,
    SLVLD       => SLVLD,
    STB_O       => STB_O,
    VA_I        => VA_I( 1 downto  1 ),
    VNAS_I      => VNAS_I,
    VNBERR_O    => VNBERR_O,
    VNDS0_I     => VNDS0_I,
    VNDS1_I     => VNDS1_I,
    VNDTACK_O   => VNDTACK_O,
    VNLWORD_I   => VNLWORD_I,
    VNWRITE_I   => VNWRITE_I,
    WE_O        => WE_O
);
```

Note: lines above the U2 block (SLALD, SLD16EO, SLDLD, SLVLD, VLOAD) appear as part of a preceding port map:

```vhdl
    SLALD       => SLALD,
    SLD16EO     => SLD16EO,
    SLDLD       => SLDLD,
    SLVLD       => SLVLD,
    VLOAD       => VLOAD
);
```

```vhdl
U5: component SLAVDECC
port map(
        A24SGL_O        => A24SGL_O,
        ADC             => ADC,
        ALOAD           => ALOAD,
        CLK_I           => CLK_I,
        SAC24_I         => SAC24_I( 23 downto 19 ),
        SAE24_I         => SAE24_I,
        VAM_I           => VAM_I( 5 downto 0 ),
        VA_I            => VA_I( 23 downto 19 ),
        VNIACK_I        => VNIACK_I
        );

U6: component VMEBADBC
port map(
        A               => A( 1 downto 1 ),
        CLK_I           => CLK_I,
        D16EO           => D16EO,
        DAT_I           => DAT_I( 31 downto 0 ),
        VD_O            => VD_O( 31 downto 0 ),
        VLOAD           => VLOAD
        );

end architecture VMECOREC1;
```

```vhdl
-- ------------------------------------------------------------------------
--
-- File name:        VmeCoreT.vhd
--
-- Description:      Test bench for the VmeCoreC entity/architecture.
--
-- Created on:       Thu Feb 14 10:03:06 2002
--
-- Source version:   VMEcore(tm) VER: 1.0.0
--
-- Interface:
--
--      Type:                     Attribute(s):
--      -----                     -------------
--      LOCAL:    LASIZE:19 LDSIZE:32
--      SLAVE:    A24:SGL D32 D16(EO) D08(O)
--      SLAVE:    BERR* SIGNAL SUPPORTED
--
```

```vhdl
-- Load the IEEE 1164 library and make it visible.
-- ------------------------------------------------------------------------

library ieee;
use ieee.std_logic_1164.all;
use std.textio.all;

-- ------------------------------------------------------------------------
-- Entity / port definition.
-- ------------------------------------------------------------------------

entity VMECORET is
end entity VMECORET;

-- ------------------------------------------------------------------------
-- Architecture definition.
-- ------------------------------------------------------------------------

architecture VMECORET1 of VMECORET is
```

```vhdl
-----------------------------------------------------------------------
-- Component definition(s).
-----------------------------------------------------------------------

component VMECOREC
port(
        A24SGL_O:       out std_logic;
        ACK_I:          in  std_logic;
        ADR_O:          out std_logic_vector( 18 downto  2 );
        CLK_I:          in  std_logic;
        CYC_O:          out std_logic;
        DAT_I:          in  std_logic_vector( 31 downto  0 );
        DAT_O:          out std_logic_vector( 31 downto  0 );
        ERR_I:          in  std_logic;
        RST_I:          in  std_logic;
        SAC24_I:        in  std_logic_vector( 23 downto 19 );
        SAE24_I:        in  std_logic;
        SEL_O:          out std_logic_vector(  3 downto  0 );
        STB_O:          out std_logic;
        VAM_I:          in  std_logic_vector(  5 downto  0 );
        VA_I:           in  std_logic_vector( 23 downto  1 );
        VD_I:           in  std_logic_vector( 31 downto  0 );
        VD_O:           out std_logic_vector( 31 downto  0 );
        VNAS_I:         in  std_logic;
        VNBERR_O:       out std_logic;
        VNDS0_I:        in  std_logic;
        VNDS1_I:        in  std_logic;
        VNDTACK_O:      out std_logic;
        VNIACK_I:       in  std_logic;
        VNLWORD_I:      in  std_logic;
        VNWRITE_I:      in  std_logic;
        WE_O:           out std_logic
    );
end component VMECOREC;

-----------------------------------------------------------------------
-- Signal definition(s).
-----------------------------------------------------------------------

signal  A24SGL_O:       std_logic;
signal  ACK_I:          std_logic;
signal  ADR_O:          std_logic_vector( 18 downto  2 );
signal  CLK_I:          std_logic;
signal  CYC_O:          std_logic;
signal  DAT_I:          std_logic_vector( 31 downto  0 );
signal  DAT_O:          std_logic_vector( 31 downto  0 );
signal  ERR_I:          std_logic;
signal  INP_VAL:        std_logic_vector( 63 downto  0 );
signal  RST_I:          std_logic;
```

```vhdl
signal SAC24_I:      std_logic_vector( 23 downto 19 );
signal SAE24_I:      std_logic;
signal SEL_O:        std_logic_vector(  3 downto  0 );
signal STB_O:        std_logic;
signal VAM_I:        std_logic_vector(  5 downto  0 );
signal VA_I:         std_logic_vector( 23 downto  1 );
signal VD_I:         std_logic_vector( 31 downto  0 );
signal VD_O:         std_logic_vector( 31 downto  0 );
signal VNAS_I:       std_logic;
signal VNBERR_O:     std_logic;
signal VNDS0_I:      std_logic;
signal VNDS1_I:      std_logic;
signal VNDTACK_O:    std_logic;
signal VNIACK_I:     std_logic;
signal VNLWORD_I:    std_logic;
signal VNWRITE_I:    std_logic;
signal WE_O:         std_logic;

begin

-- -----------------------------------------------------
-- Component map.
-- -----------------------------------------------------

U1: component VMECOREC
port map (

    A24SGL_O    => A24SGL_O,
    ACK_I       => ACK_I,
    ADR_O       => ADR_O( 18 downto  2 ),
    CLK_I       => CLK_I,
    CYC_O       => CYC_O,
    DAT_I       => DAT_I( 31 downto  0 ),
    DAT_O       => DAT_O( 31 downto  0 ),
    ERR_I       => ERR_I,
    RST_I       => RST_I,
    SAC24_I     => SAC24_I( 23 downto 19 ),
    SAE24_I     => SAE24_I,
    SEL_O       => SEL_O(  3 downto  0 ),
    STB_O       => STB_O,
    VAM_I       => VAM_I(  5 downto  0 ),
    VA_I        => VA_I( 23 downto  1 ),
    VD_I        => VD_I( 31 downto  0 ),
    VD_O        => VD_O( 31 downto  0 ),
    VNAS_I      => VNAS_I,
    VNBERR_O    => VNBERR_O,
    VNDS0_I     => VNDS0_I,
    VNDS1_I     => VNDS1_I,
    VNDTACK_O   => VNDTACK_O,
    VNIACK_I    => VNIACK_I,
```

```vhdl
        VNLWORD_I  => VNLWORD_I,
        VNWRITE_I  => VNWRITE_I,
        WE_O       => WE_O

    );

-------------------------------------------------------------------
-- Test process.
-------------------------------------------------------------------

VMECORE_TEST: process

-------------------------------------------------------------------
-- Define file parameters.
-------------------------------------------------------------------

variable   C:          character;
variable   GOOD:       boolean;
variable   L:          line;
variable   LINENUM:    string(3 downto 0);
file       TVF:        text;
variable   UNKNOWN:    boolean;

-------------------------------------------------------------------
-- Define constants.
-------------------------------------------------------------------

constant DLY: time := 5 ns;

-------------------------------------------------------------------
-- Procedure READHEX.
-------------------------------------------------------------------

procedure READHEX is
begin

    wait for 0 fs;
    INP_VAL <= X"000000000000000000";
    UNKNOWN := false;
    read(L, C, GOOD); assert(GOOD) report "TEXT I/O READ ERROR" severity ERROR;
    while(C /= ' ') loop
        wait for 0 fs;
        INP_VAL(63 downto  4) <= INP_VAL(59 downto 0);
        wait for 0 fs;
        case(C) is
            when '0' => INP_VAL(3 downto 0)  <= B"0000";
            when '1' => INP_VAL(3 downto 0)  <= B"0001";
            when '2' => INP_VAL(3 downto 0)  <= B"0010";
```

```vhdl
          when '3' =>  INP_VAL(3 downto 0) <= B"0011";
          when '4' =>  INP_VAL(3 downto 0) <= B"0100";
          when '5' =>  INP_VAL(3 downto 0) <= B"0101";
          when '6' =>  INP_VAL(3 downto 0) <= B"0110";
          when '7' =>  INP_VAL(3 downto 0) <= B"0111";
          when '8' =>  INP_VAL(3 downto 0) <= B"1000";
          when '9' =>  INP_VAL(3 downto 0) <= B"1001";
          when 'A' =>  INP_VAL(3 downto 0) <= B"1010";
          when 'B' =>  INP_VAL(3 downto 0) <= B"1011";
          when 'C' =>  INP_VAL(3 downto 0) <= B"1100";
          when 'D' =>  INP_VAL(3 downto 0) <= B"1101";
          when 'E' =>  INP_VAL(3 downto 0) <= B"1110";
          when 'F' =>  INP_VAL(3 downto 0) <= B"1111";
          when 'X' =>  UNKNOWN := true;
          when others => report "UNRECOGNIZED TEST VECTOR INPUT ON LINE 0x" & LINENUM severity ERROR;
        end case;
      read(L, C, GOOD); assert(GOOD) report "TEXT I/O READ ERROR" severity ERROR;
    end loop;
    wait for 0 fs;

end procedure READHEX;

begin

  -- Open the test vector file.
  ------------------------------------------------

report "RUNNING TEST VECTORS IN FILE VMECOREV.TXT" severity NOTE;
file_open( TVF, "VMECOREV.TXT", read_mode );

  ------------------------------------------------
  -- Read each test vector from the file and apply it.
  ------------------------------------------------

APPLY_VECTORS: loop

  ------------------------------------------------
  -- Exit at the end of the file.
  ------------------------------------------------

  if endfile(TVF) then
    file_close(TVF);
    report "TEST VECTORS COMPLETE." severity NOTE;
    exit;
  end if;
```

```vhdl
            -----------------------------------------------------
            -- Read the next line in the test vector file.
            -----------------------------------------------------

            readline(TVF, L );


            -----------------------------------------------------
            -- Ignore empty lines.
            -----------------------------------------------------

            next when(L'length = 0 );


            -----------------------------------------------------
            -- Ignore non-test vector lines.
            -----------------------------------------------------

            read(L, C, GOOD);
            assert(GOOD) report "TEXT I/O READ ERROR" severity ERROR;
            next when( C /= ':' );


            -----------------------------------------------------
            -- Read the test vector number.
            -----------------------------------------------------

            read(L, C, GOOD);
            assert(GOOD) report "TEXT I/O READ ERROR" severity ERROR;
            read(L,LINENUM(3), GOOD);
            assert(GOOD) report "TEXT I/O READ ERROR" severity ERROR;
            read(L,LINENUM(2), GOOD);
            assert(GOOD) report "TEXT I/O READ ERROR" severity ERROR;
            read(L,LINENUM(1), GOOD);
            assert(GOOD) report "TEXT I/O READ ERROR" severity ERROR;
            read(L,LINENUM(0), GOOD);
            assert(GOOD) report "TEXT I/O READ ERROR" severity ERROR;
            read(L, C, GOOD); assert(GOOD) report "TEXT I/O READ ERROR" severity ERROR;


            -----------------------------------------------------
            -- Apply the test vector inputs.
            -----------------------------------------------------

            wait for DLY; CLK_I <= '0'; wait for DLY;

            read(L, C, GOOD); assert(GOOD) report "TEXT I/O READ ERROR" severity ERROR;
            if   (C = '0') then ACK_I <= '0';
            elsif(C = '1') then ACK_I <= '1';
            else report "UNRECOGNIZED TEST VECTOR INPUT FOR ACK_I ON LINE 0x" & LINENUM severity ERROR;
```

```vhdl
end if;
read(L, C, GOOD); assert(GOOD) report "TEXT I/O READ ERROR" severity ERROR;
if(C /= ' ') then report "WHITESPACE EXPECTED IN TEST VECTOR ON LINE 0x" & LINENUM severity ERROR; end if;

READHEX;
DAT_I(31 downto 0) <= INP_VAL(31 downto 0);

read(L, C, GOOD); assert(GOOD) report "TEXT I/O READ ERROR" severity ERROR;
if   (C = '0') then ERR_I <= '0';
elsif(C = '1') then ERR_I <= '1';
else report "UNRECOGNIZED TEST VECTOR INPUT FOR ERR_I ON LINE 0x" & LINENUM severity ERROR;
end if;
read(L, C, GOOD); assert(GOOD) report "TEXT I/O READ ERROR" severity ERROR; end if;
if(C /= ' ') then report "WHITESPACE EXPECTED IN TEST VECTOR ON LINE 0x" & LINENUM severity ERROR;

read(L, C, GOOD); assert(GOOD) report "TEXT I/O READ ERROR" severity ERROR;
if   (C = '0') then RST_I <= '0';
elsif(C = '1') then RST_I <= '1';
else report "UNRECOGNIZED TEST VECTOR INPUT FOR RST_I ON LINE 0x" & LINENUM severity ERROR;
end if;
read(L, C, GOOD); assert(GOOD) report "TEXT I/O READ ERROR" severity ERROR; end if;
if(C /= ' ') then report "WHITESPACE EXPECTED IN TEST VECTOR ON LINE 0x" & LINENUM severity ERROR;

READHEX;
SAC24_I(23 downto 19) <= INP_VAL(4 downto 0);

read(L, C, GOOD); assert(GOOD) report "TEXT I/O READ ERROR" severity ERROR;
if   (C = '0') then SAE24_I <= '0';
elsif(C = '1') then SAE24_I <= '1';
else report "UNRECOGNIZED TEST VECTOR INPUT FOR SAE24_I ON LINE 0x" & LINENUM severity ERROR;
else report "UNRECOGNIZED TEST VECTOR INPUT FOR SAE24_I ON LINE 0x" & LINENUM severity ERROR;
read(L, C, GOOD); assert(GOOD) report "TEXT I/O READ ERROR" severity ERROR; end if;
if(C /= ' ') then report "WHITESPACE EXPECTED IN TEST VECTOR ON LINE 0x" & LINENUM severity ERROR; end if;

READHEX;
VAM_I(5 downto 0) <= INP_VAL(5 downto 0);

READHEX;
VA_I(23 downto 1) <= INP_VAL(22 downto 0);

READHEX;
VD_I(31 downto 0) <= INP_VAL(31 downto 0);

read(L, C, GOOD); assert(GOOD) report "TEXT I/O READ ERROR" severity ERROR;
if   (C = '0') then VNAS_I <= '0';
elsif(C = '1') then VNAS_I <= '1';
else report "UNRECOGNIZED TEST VECTOR INPUT FOR VNAS_I ON LINE 0x" & LINENUM severity ERROR;
end if;
read(L, C, GOOD); assert(GOOD) report "TEXT I/O READ ERROR" severity ERROR; end if;
if(C /= ' ') then report "WHITESPACE EXPECTED IN TEST VECTOR ON LINE 0x" & LINENUM severity ERROR;
```

```vhdl
read(L, C, GOOD); assert(GOOD) report "TEXT I/O READ ERROR" severity ERROR;
if   (C = '0') then VNDS0_I <= '0';
elsif(C = '1') then VNDS0_I <= '1';
else report "UNRECOGNIZED TEST VECTOR INPUT FOR VNDS0_I ON LINE 0x" & LINENUM severity ERROR;
end if;
read(L, C, GOOD); assert(GOOD) report "TEXT I/O READ ERROR" severity ERROR;
if(C /= ' ') then report "WHITESPACE EXPECTED IN TEST VECTOR ON LINE 0x" & LINENUM severity ERROR; end if;

read(L, C, GOOD); assert(GOOD) report "TEXT I/O READ ERROR" severity ERROR;
if   (C = '0') then VNDS1_I <= '0';
elsif(C = '1') then VNDS1_I <= '1';
else report "UNRECOGNIZED TEST VECTOR INPUT FOR VNDS1_I ON LINE 0x" & LINENUM severity ERROR;
end if;
read(L, C, GOOD); assert(GOOD) report "TEXT I/O READ ERROR" severity ERROR;
if(C /= ' ') then report "WHITESPACE EXPECTED IN TEST VECTOR ON LINE 0x" & LINENUM severity ERROR; end if;

read(L, C, GOOD); assert(GOOD) report "TEXT I/O READ ERROR" severity ERROR;
if   (C = '0') then VNIACK_I <= '0';
elsif(C = '1') then VNIACK_I <= '1';
else report "UNRECOGNIZED TEST VECTOR INPUT FOR VNIACK_I ON LINE 0x" & LINENUM severity ERROR;
end if;
read(L, C, GOOD); assert(GOOD) report "TEXT I/O READ ERROR" severity ERROR;
if(C /= ' ') then report "WHITESPACE EXPECTED IN TEST VECTOR ON LINE 0x" & LINENUM severity ERROR; end if;

read(L, C, GOOD); assert(GOOD) report "TEXT I/O READ ERROR" severity ERROR;
if   (C = '0') then VNLWORD_I <= '0';
elsif(C = '1') then VNLWORD_I <= '1';
else report "UNRECOGNIZED TEST VECTOR INPUT FOR VNLWORD_I ON LINE 0x" & LINENUM severity ERROR;
end if;
read(L, C, GOOD); assert(GOOD) report "TEXT I/O READ ERROR" severity ERROR;
if(C /= ' ') then report "WHITESPACE EXPECTED IN TEST VECTOR ON LINE 0x" & LINENUM severity ERROR; end if;

read(L, C, GOOD); assert(GOOD) report "TEXT I/O READ ERROR" severity ERROR;
if   (C = '0') then VNWRITE_I <= '0';
elsif(C = '1') then VNWRITE_I <= '1';
else report "UNRECOGNIZED TEST VECTOR INPUT FOR VNWRITE_I ON LINE 0x" & LINENUM severity ERROR;
end if;
read(L, C, GOOD); assert(GOOD) report "TEXT I/O READ ERROR" severity ERROR;
if(C /= ' ') then report "WHITESPACE EXPECTED IN TEST VECTOR ON LINE 0x" & LINENUM severity ERROR; end if;

-------------------------------------------------
-- Clock input.
-------------------------------------------------

read(L, C, GOOD); assert(GOOD) report "TEXT I/O READ ERROR" severity ERROR;
if(C = 'C') then wait for DLY; CLK_I <= '1'; wait for DLY;
else report "TEST VECTOR ERROR ON CLK_I INPUT ON LINE 0x" & LINENUM severity ERROR;
end if;
```

```vhdl
read(L, C, GOOD); assert(GOOD) report "TEXT I/O READ ERROR" severity ERROR;
while(C /= ' ') loop
    read(L, C, GOOD); assert(GOOD) report "TEXT I/O READ ERROR" severity ERROR;
end loop;

-------------------------------------------------------------------
-- Check the test vector outputs.
-------------------------------------------------------------------

read(L, C, GOOD); assert(GOOD) report "TEXT I/O READ ERROR" severity ERROR;
if    (C = '0') then assert(A24SGL_O = '0')
    report "TEST VECTOR ERROR ON OUTPUT A24SGL_O = '0') ON LINE 0x" & LINENUM severity ERROR;
elsif(C = '1') then assert(A24SGL_O = '1')
    report "TEST VECTOR ERROR ON OUTPUT A24SGL_O = '1')
elsif(C /= 'x') then
    report "UNRECOGNIZED TEST VECTOR OUTPUT FOR A24SGL_O ON LINE 0x" & LINENUM severity ERROR;
end if;
read(L, C, GOOD); assert(GOOD) report "TEXT I/O READ ERROR" severity ERROR;
if(C /= ' ') then report "WHITESPACE EXPECTED IN TEST VECTOR ON LINE 0x" & LINENUM severity ERROR; end if;

end if;

READHEX;
if(UNKNOWN = false) then
    assert(ADR_O(18 downto 2) = INP_VAL(16 downto 0))
    report "TEST VECTOR ERROR ON OUTPUT ADR_O ON LINE 0x" & LINENUM severity ERROR;

read(L, C, GOOD); assert(GOOD) report "TEXT I/O READ ERROR" severity ERROR;
if    (C = '0') then assert(CYC_O = '0')
    report "TEST VECTOR ERROR ON OUTPUT CYC_O ON LINE 0x" & LINENUM severity ERROR;
elsif(C = '1') then assert(CYC_O = '1')
    report "TEST VECTOR ERROR ON OUTPUT CYC_O ON LINE 0x" & LINENUM severity ERROR;
elsif(C /= 'x') then
    report "UNRECOGNIZED TEST VECTOR OUTPUT FOR CYC_O ON LINE 0x" & LINENUM severity ERROR;
end if;
read(L, C, GOOD); report "TEXT I/O READ ERROR" severity ERROR;
if(C /= ' ') then report "WHITESPACE EXPECTED IN TEST VECTOR ON LINE 0x" & LINENUM severity ERROR;

end if;

READHEX;
if(UNKNOWN = false) then
    assert(DAT_O(31 downto 0) = INP_VAL(31 downto 0))
    report "TEST VECTOR ERROR ON OUTPUT DAT_O ON LINE 0x" & LINENUM severity ERROR;

READHEX;
if(UNKNOWN = false) then
    assert(SEL_O(3 downto 0) = INP_VAL(3 downto 0))
    report "TEST VECTOR ERROR ON OUTPUT SEL_O ON LINE 0x" & LINENUM severity ERROR;

end if;
```

```vhdl
    read(L, C, GOOD); assert(GOOD) report "TEXT I/O READ ERROR" severity ERROR;
    if   (C = '0') then assert(STB_O = '0')
        report "TEST VECTOR ERROR ON OUTPUT STB_O ON LINE 0x" & LINENUM severity ERROR;
    elsif(C = '1') then assert(STB_O = '1')
        report "TEST VECTOR ERROR ON OUTPUT STB_O ON LINE 0x" & LINENUM severity ERROR;
    elsif(C /= 'x') then
        report "UNRECOGNIZED TEST VECTOR OUTPUT FOR STB_O ON LINE 0x" & LINENUM severity ERROR;
    end if;
    read(L, C, GOOD); assert(GOOD) report "TEXT I/O READ ERROR" severity ERROR;
    if(C /= ' ') then report "WHITESPACE EXPECTED IN TEST VECTOR ON LINE 0x" & LINENUM severity ERROR; end if;

READHEX;
    if(UNKNOWN = false) then
        assert(VD_O(31 downto 0) = INP_VAL(31 downto 0))
            report "TEST VECTOR ERROR ON OUTPUT VD_O ON LINE 0x" & LINENUM severity ERROR;
    end if;

    read(L, C, GOOD); assert(GOOD) report "TEXT I/O READ ERROR" severity ERROR;
    if   (C = '0') then assert(VNBERR_O = '0')
        report "TEST VECTOR ERROR ON OUTPUT VNBERR_O ON LINE 0x" & LINENUM severity ERROR;
    elsif(C = '1') then assert(VNBERR_O = '1')
        report "TEST VECTOR ERROR ON OUTPUT VNBERR_O ON LINE 0x" & LINENUM severity ERROR;
    elsif(C /= 'x') then
        report "UNRECOGNIZED TEST VECTOR OUTPUT FOR VNBERR_O ON LINE 0x" & LINENUM severity ERROR;
    end if;
    read(L, C, GOOD); assert(GOOD) report "TEXT I/O READ ERROR" severity ERROR;
    if(C /= ' ') then report "WHITESPACE EXPECTED IN TEST VECTOR ON LINE 0x" & LINENUM severity ERROR; end if;

    read(L, C, GOOD); assert(GOOD) report "TEXT I/O READ ERROR" severity ERROR;
    if   (C = '0') then assert(VNDTACK_O = '0')
        report "TEST VECTOR ERROR ON OUTPUT VNDTACK_O ON LINE 0x" & LINENUM severity ERROR;
    elsif(C = '1') then assert(VNDTACK_O = '1')
        report "TEST VECTOR ERROR ON OUTPUT VNDTACK_O ON LINE 0x" & LINENUM severity ERROR;
    elsif(C /= 'x') then
        report "UNRECOGNIZED TEST VECTOR OUTPUT FOR VNDTACK_O ON LINE 0x" & LINENUM severity ERROR;
    end if;
    read(L, C, GOOD); assert(GOOD) report "TEXT I/O READ ERROR" severity ERROR;
    if(C /= ' ') then report "WHITESPACE EXPECTED IN TEST VECTOR ON LINE 0x" & LINENUM severity ERROR; end if;

    read(L, C, GOOD); assert(GOOD) report "TEXT I/O READ ERROR" severity ERROR;
    if   (C = '0') then assert(WE_O = '0')
        report "TEST VECTOR ERROR ON OUTPUT WE_O ON LINE 0x" & LINENUM severity ERROR;
    elsif(C = '1') then assert(WE_O = '1')
        report "TEST VECTOR ERROR ON OUTPUT WE_O ON LINE 0x" & LINENUM severity ERROR;
    elsif(C /= 'x') then
        report "UNRECOGNIZED TEST VECTOR OUTPUT FOR WE_O ON LINE 0x" & LINENUM severity ERROR;
    end if;
    read(L, C, GOOD); assert(GOOD) report "TEXT I/O READ ERROR" severity ERROR;
    if(C /= ' ') then report "WHITESPACE EXPECTED IN TEST VECTOR ON LINE 0x" & LINENUM severity ERROR; end if;
```

```vhdl
        end loop APPLY_VECTORS;

        ----------------------------------------
        -- All done.
        ----------------------------------------

        wait;

    end process VMECORE_TEST;

end architecture VMECORET1;
```

```
-- --------------------------------------------------------------------
-- File name:        VmeCoreV.txt
--
-- Description:      Test vectors for the VmeCoreT test bench.
--
-- Created on:       Thu Feb 14 10:03:06 2002
--
-- Source version:   VMEcore(tm) VER: 1.0.0
--
-- Interface:        Type:            Attribute(s):
--                   ------           -------------
--                   LOCAL:   LASIZE:19 LDSIZE:32
--                   SLAVE:   A24:SGL D32 D16(EO) D08(O)
--                   SLAVE:   BERR* SIGNAL SUPPORTED
--
```

```
-- --------------------------------------------------------------------
--
-- Test vectors:   ACK_I
--                 DAT_I             ( 31 downto  0 )
--                 ERR_I
--                 RST_I
--                 SAC24_I           ( 23 downto 19 )
--                 SAE24_I
--                 VAM_I             (  5 downto  0 )
--                 VA_I              ( 23 downto  1 )
--                 VD_I              ( 31 downto  0 )
--                 VNAS_I
--                 VNDS0_I
--                 VNDS1_I
--                 VNIACK_I
--                 VNLWORD_I
--                 VNWRITE_I
--                 CLK_I
--                 A24SGL_O
--                 ADR_O             ( 18 downto  2 )
--                 CYC_O
--                 DAT_O             ( 31 downto  0 )
--                 SEL_O             (  3 downto  0 )
--                 STB_O
```

```
                                                    VD_O   ( 31 downto  0 )
                                                    VNBERR_O
                                                    VNDTACK_O
                                                    WE_O
```

-- Apply WISHBONE reset, and verify the initial output conditions.

```
0000 0 00000000 0 1 00 0 3F 7FFFFF FFFFFFFF 1 1 1 1 1 CLK X XXXXX X XXXXXXXX X X XXXXXXXX X 1 1 X
0001 0 00000000 0 1 00 0 3F 7FFFFF FFFFFFFF 1 1 1 1 1 CLK X XXXXX X XXXXXXXX X X XXXXXXXX X 1 1 X
0002 0 00000000 0 0 00 0 3F 7FFFFF FFFFFFFF 1 1 1 1 1 CLK X XXXXX X XXXXXXXX X X XXXXXXXX X 1 1 X
0003 0 00000000 0 0 00 0 3F 7FFFFF FFFFFFFF 1 1 1 1 1 CLK X XXXXX X XXXXXXXX X X XXXXXXXX X 1 1 X
0004 0 00000000 0 0 00 0 3F 7FFFFF FFFFFFFF 1 1 1 1 1 CLK X XXXXX X XXXXXXXX X X XXXXXXXX X 1 1 X
0005 0 00000000 0 0 00 0 3F 7FFFFF FFFFFFFF 1 1 1 1 1 CLK X XXXXX X XXXXXXXX X X XXXXXXXX X 1 1 X
```

-- SLAVE, A24:SGL addressing test.

```
0006 0 00000000 0 0 0A 1 3E 298421 FFFFFFFF 0 0 1 1 0 CLK X XXXXX 0 XXXXXXXX X X XXXXXXXX X 1 1 X
0007 0 00000000 0 0 0A 1 3E 298421 FFFFFFFF 0 0 1 1 0 CLK 1 0C210 1 XXXXXXXX X X XXXXXXXX X 1 1 X
0008 1 00000000 0 0 0A 1 3E 298421 FFFFFFFF 0 0 1 1 0 CLK 1 0C210 1 XXXXXXXX X X XXXXXXXX X 1 1 X
0009 0 00000000 0 0 0A 1 3E 298421 FFFFFFFF 0 0 1 1 0 CLK 1 0C210 0 XXXXXXXX X X XXXXXXXX X 1 1 X
000A 0 00000000 0 0 0A 1 3E 530842 FFFFFFFF 0 0 1 1 0 CLK 1 18421 1 XXXXXXXX X X XXXXXXXX X 1 1 X
000B 0 00000000 0 0 14 1 3E 530842 FFFFFFFF 0 0 1 1 0 CLK 1 18421 1 XXXXXXXX X X XXXXXXXX X 1 1 X
000C 1 00000000 0 0 14 1 3E 530842 FFFFFFFF 0 0 1 1 0 CLK 1 18421 1 XXXXXXXX X X XXXXXXXX X 1 1 X
000D 0 00000000 0 0 14 1 3E 530842 FFFFFFFF 1 1 1 1 0 CLK 1 18421 0 XXXXXXXX X X XXXXXXXX X 1 1 X
000E 0 00000000 0 0 09 1 3E 261084 FFFFFFFF 0 0 1 1 0 CLK 1 10842 1 XXXXXXXX X X XXXXXXXX X 1 1 X
000F 0 00000000 0 0 09 1 3E 261084 FFFFFFFF 0 0 1 1 0 CLK 1 10842 1 XXXXXXXX X X XXXXXXXX X 1 1 X
0010 1 00000000 0 0 09 1 3E 261084 FFFFFFFF 0 0 1 1 0 CLK 1 10842 1 XXXXXXXX X X XXXXXXXX X 1 1 X
0011 0 00000000 0 0 09 1 3E 261084 FFFFFFFF 0 0 1 1 0 CLK 1 10842 0 XXXXXXXX X X XXXXXXXX X 1 1 X
0012 0 00000000 0 0 3E 1 3E 4C2108 FFFFFFFF 1 1 1 1 0 CLK 1 01084 1 XXXXXXXX X X XXXXXXXX X 1 1 X
0013 0 00000000 0 0 3E 1 3E 4C2108 FFFFFFFF 0 0 1 1 0 CLK 1 01084 1 XXXXXXXX X X XXXXXXXX X 1 1 X
0014 1 00000000 0 0 3E 1 3E 4C2108 FFFFFFFF 0 0 1 1 1 CLK 1 01084 1 XXXXXXXX X X XXXXXXXX X 1 0 X
0015 0 00000000 0 0 13 1 3E 4C2108 FFFFFFFF 0 0 1 1 0 CLK 1 01084 0 XXXXXXXX X X XXXXXXXX X 1 1 X
0016 0 00000000 0 0 1F 0 3E 7FFFFF FFFFFFFF 0 0 1 1 0 CLK 1 1FFFF 0 XXXXXXXX X X XXXXXXXX X 1 1 X
0017 0 00000000 0 0 1F 0 3E 7FFFFF FFFFFFFF 1 1 1 0 1 CLK 1 1FFFF 0 XXXXXXXX X X XXXXXXXX X 1 1 X
0018 0 00000000 0 0 1F 0 3E 7FFFFF FFFFFFFF 1 1 0 0 1 CLK 1 1FFFF 0 XXXXXXXX X X XXXXXXXX X 1 1 X
0019 0 00000000 0 0 00 1 3E 7FFFFF FFFFFFFF 1 1 0 0 1 CLK 1 1FFFF 0 XXXXXXXX X X XXXXXXXX X 1 1 X
001A 0 00000000 0 0 00 1 3E 7FFFFF FFFFFFFF 1 1 0 0 1 CLK 1 1FFFF 0 XXXXXXXX X X XXXXXXXX X 1 1 X
001B 0 00000000 0 0 1F 1 3E 7FFFFF FFFFFFFF 1 1 0 0 0 CLK 1 1FFFF 0 XXXXXXXX X X XXXXXXXX X 1 1 X
```

```
.. 001C 0 00000000 0 1F 1 3D 7FFFF FFFFFFFF 0 0 1 1 1 CLK 1 1FFFF 0 XXXXXXXX X 0 XXXXXXXX 1 1 1 X
.. 001D 0 00000000 0 1F 1 3D 7FFFF FFFFFFFF 0 0 1 1 1 CLK 1 1FFFF 1 XXXXXXXX X 1 XXXXXXXX 1 1 0 X
.. 001E 1 00000000 0 1F 1 3D 7FFFF FFFFFFFF 0 0 1 1 1 CLK 1 1FFFF 1 XXXXXXXX X 0 XXXXXXXX 1 0 1 X
.. 001F 0 00000000 0 1F 1 3D 7FFFF FFFFFFFF 1 0 1 1 1 CLK 1 1FFFF 0 XXXXXXXX X 0 XXXXXXXX 1 1 0 X
.. 0020 0 00000000 0 1F 1 3A 7FFFF FFFFFFFF 0 0 1 1 1 CLK 1 1FFFF 1 XXXXXXXX X 1 XXXXXXXX 1 1 1 X
.. 0021 1 00000000 0 1F 1 3A 7FFFF FFFFFFFF 0 0 1 1 1 CLK 1 1FFFF 0 XXXXXXXX X 0 XXXXXXXX 1 1 0 X
.. 0022 0 00000000 0 1F 1 3A 7FFFF FFFFFFFF 1 0 1 1 1 CLK 1 1FFFF 1 XXXXXXXX X 0 XXXXXXXX 1 0 1 X
.. 0023 1 00000000 0 1F 1 3A 7FFFF FFFFFFFF 0 0 1 1 1 CLK 1 1FFFF 0 XXXXXXXX X 0 XXXXXXXX 1 1 0 X
.. 0024 0 00000000 0 1F 1 39 7FFFF FFFFFFFF 0 0 1 1 1 CLK 1 1FFFF 1 XXXXXXXX X 1 XXXXXXXX 1 1 1 X
.. 0025 1 00000000 0 1F 1 39 7FFFF FFFFFFFF 0 0 1 1 1 CLK 1 1FFFF 0 XXXXXXXX X 0 XXXXXXXX 1 1 0 X
.. 0026 1 00000000 0 1F 1 39 7FFFF FFFFFFFF 0 1 1 1 1 CLK 1 1FFFF 1 XXXXXXXX X 0 XXXXXXXX 1 0 1 X
.. 0027 0 00000000 0 1F 1 1F 7FFFF FFFFFFFF 0 0 1 1 1 CLK 1 1FFFF 0 XXXXXXXX X 0 XXXXXXXX 1 1 1 X
.. 0028 0 00000000 0 1F 1 1F 7FFFF FFFFFFFF 1 0 1 1 1 CLK 1 1FFFF 1 XXXXXXXX X 0 XXXXXXXX 1 1 1 X
.. 0029 0 00000000 0 1F 1 1F 7FFFF FFFFFFFF 0 0 1 1 0 CLK 1 1FFFF 1 XXXXXXXX X 0 XXXXXXXX 1 1 1 X
.. 002A 0 00000000 0 1F 1 3E 7FFFF FFFFFFFF 0 0 1 1 1 CLK 1 1FFFF 0 XXXXXXXX X 0 XXXXXXXX 1 1 1 X
.. 002B 0 00000000 0 1F 1 3E 7FFFF FFFFFFFF 1 0 1 0 1 CLK 1 1FFFF 0 XXXXXXXX X 0 XXXXXXXX 1 1 1 X
.. 002C 0 00000000 0 1F 1 3E 7FFFF FFFFFFFF 0 0 1 1 1 CLK 1 1FFFF 0 XXXXXXXX X 0 XXXXXXXX 1 1 1 X
.. 002D 1 00000000 0 1F 1 3E 7FFFF FFFFFFFF 1 0 1 1 1 CLK 0 1FFFF 0 XXXXXXXX X 1 XXXXXXXX 1 1 1 X
.. 002E 0 00000000 0 1F 1 3E 7FFFF FFFFFFFF 0 0 1 1 1 CLK 0 1FFFF 0 XXXXXXXX X 0 XXXXXXXX 1 0 1 X
.. 002F 0 00000000 0 1F 1 3E 7FFFF FFFFFFFF 0 1 1 1 1 CLK 0 1FFFF 0 XXXXXXXX X 0 XXXXXXXX 1 1 1 X
.. 0030 0 00000000 0 1F 1 3E 7FFFF FFFFFFFF 0 0 1 1 1 CLK 0 1FFFF 0 XXXXXXXX X 0 XXXXXXXX 1 1 1 X
.. 0031 1 00000000 0 1F 1 3E 7FFFF FFFFFFFF 0 1 1 1 1 CLK 1 1FFFF 1 XXXXXXXX X 1 XXXXXXXX 1 1 1 X
.. 0032 1 00000000 0 1F 1 3E 7FFFF FFFFFFFF 0 0 1 1 0 CLK 1 1FFFF 0 XXXXXXXX X 0 XXXXXXXX 1 0 1 X
.. 0033 1 00000000 0 1F 1 3E 7FFFF FFFFFFFF 0 0 1 1 1 CLK 1 1FFFF 1 XXXXXXXX X 1 XXXXXXXX 1 1 1 X
.. 0034 0 00000000 0 1F 1 3E 7FFFF FFFFFFFF 1 1 1 1 1 CLK 1 1FFFF 0 XXXXXXXX X 0 XXXXXXXX 1 1 1 X

-- SLAVE, D16 (EO) data interface test.

.. 0035 1 00000000 0 1F 1 3E 7FFFFE 00000000 0 0 1 1 1 CLK 1 1FFFF 0 XXXXXXXX X 0 XXXXXXXX 1 1 1 X
.. 0036 1 00000000 0 1F 1 3E 7FFFFE 00000000 0 0 1 1 1 CLK 1 1FFFF 1 XXXXXXXX X 4 1 XXXXXXXX 1 1 0 X
.. 0037 1 00A90000 0 1F 1 3E 7FFFFE 00000000 0 0 1 1 1 CLK 1 1FFFF 1 00A900A9 0 0 00A900A9 1 1 0 0
.. 0038 1 00000000 0 1F 1 3E 7FFFFE 00000000 0 0 1 1 1 CLK 1 1FFFF 1 00A900A9 0 0 00A900A9 1 1 0 0
.. 0039 1 00000000 0 1F 1 3E 7FFFFE 00000000 0 1 1 1 1 CLK 1 1FFFF 1 XXXXXXXX 0 0 XXXXXXXX 1 1 0 0
.. 003A 1 00000000 0 1F 1 3E 7FFFFE 00000021 0 0 1 1 0 CLK 1 1FFFF 1 XXXXXXXX 0 0 XXXXXXXX 1 1 0 1
.. 003B 1 00000000 0 1F 1 3E 7FFFFE 00000021 0 0 1 0 1 CLK 0 1FFFF 1 00210021 4 1 00210021 1 1 1 1
.. 003C 1 00000000 0 1F 1 3E 7FFFFE 00000000 1 1 0 1 0 CLK 0 1FFFF 0 00210021 0 0 00210021 1 1 1 1
.. 003D 1 00000000 0 1F 1 3E 7FFFFE 00000000 0 1 1 0 0 CLK 1 1FFFF 1 XXXXXXXX 0 0 XXXXXXXX 1 1 1 1
.. 003E 1 00000000 0 1F 1 3E 7FFFFE 00000000 0 0 1 1 1 CLK 1 1FFFF 1 XXXXXXXX 8 1 XXXXXXXX 1 1 0 0
.. 003F 1 6C000000 0 1F 1 3E 7FFFFE 6C000000 0 1 0 0 1 CLK 1 1FFFF 1 6C006C00 0 0 6C006C00 1 1 0 0
.. 0040 1 00000000 0 1F 1 3E 7FFFFE 00000000 0 1 1 0 1 CLK 1 1FFFF 1 6C006C00 0 0 6C006C00 1 1 0 0
.. 0041 1 00000000 0 1F 1 3E 7FFFFE 00000000 0 1 1 1 0 CLK 0 1FFFF 0 XXXXXXXX 0 0 XXXXXXXX 1 1 0 1
.. 0042 1 00008400 0 1F 1 3E 7FFFFE 00008400 0 1 0 0 1 CLK 1 1FFFF 1 84008400 8 1 84008400 1 1 1 1
.. 0043 1 00008400 0 1F 1 3E 7FFFFE 00008400 1 1 0 0 0 CLK 1 1FFFF 0 84008400 0 0 84008400 1 1 1 1
.. 0044 1 00000000 0 1F 1 3E 7FFFFE 00000000 1 1 1 0 0 CLK 0 1FFFF 1 XXXXXXXX 0 0 XXXXXXXX 1 1 0 0
.. 0045 1 00000000 0 1F 1 3E 7FFFFE 00000000 0 1 1 1 1 CLK 1 1FFFF 1 XXXXXXXX C 1 XXXXXXXX 1 1 0 1
.. 0046 1 00000000 0 1F 1 3E 7FFFFE 00000000 0 0 1 1 1 CLK 1 1FFFF 1 XXXXXXXX 0 0 XXXXXXXX 1 1 1 1
.. 0047 1 6CA90000 0 1F 1 3E 7FFFFE 6CA90000 0 0 1 1 0 CLK 1 1FFFF 1 XXXXXXXX 0 0 6CA96CA9 1 1 0 0
.. 0048 1 00000000 0 1F 1 3E 7FFFFE 00000000 0 0 1 1 1 CLK 1 1FFFF 1 XXXXXXXX 0 0 6CA96CA9 1 0 0 0
.. 0049 0 00000000 0 1F 1 3E 7FFFFE 00000000 1 0 1 0 1 CLK 1 1FFFF 0 XXXXXXXX 0 0 6CA96CA9 1 0 0 0
.. 004A 0 00000000 0 1F 1 3E 7FFFFE 00000000 0 1 1 0 1 CLK 1 1FFFF 0 XXXXXXXX 0 0 6CA96CA9 1 0 0 0
```

Addr																	
004B	0	00000000	0	1F	1	3E	7FFFFE	00000000	0	1	CLK	1	1FFFF	0	XXXXXXXX	0	0
004C	0	00000000	0	1F	1	3E	7FFFFE	00000000	0	1	CLK	1	1FFFF	0	XXXXXXXX	0	0
004D	0	00000000	0	1F	1	3E	7FFFFE	00008421	0	1	CLK	1	1FFFF	0	84218421	C	1
004E	0	00000000	0	1F	1	3E	7FFFFE	00008421	0	1	CLK	1	1FFFF	0	84218421	C	1
004F	1	00000000	0	1F	1	3E	7FFFFE	00000000	1	0	CLK	1	1FFFF	0	XXXXXXXX	0	0
0050	1	00000000	0	1F	1	3E	7FFFFE	00000000	1	0	CLK	1	1FFFF	0	XXXXXXXX	0	0
0051	1	00530000	0	1F	1	3E	7FFFFE	00000000	1	0	CLK	1	1FFFF	1	00530053	0	0
0052	1	00000000	0	1F	1	3E	7FFFFE	00000000	1	0	CLK	1	1FFFF	1	00530053	0	0
0053	1	00000000	0	1F	1	3E	7FFFFE	00000000	1	0	CLK	1	1FFFF	1	00530053	0	0
0054	0	00000000	0	1F	1	3E	7FFFFE	00000000	1	1	CLK	1	1FFFF	1	XXXXXXXX	0	0
0055	0	00000000	0	1F	1	3E	7FFFFE	00000000	1	1	CLK	1	1FFFF	1	XXXXXXXX	0	0
0056	0	00000000	0	1F	1	3E	7FFFFE	00000000	1	1	CLK	1	1FFFF	1	XXXXXXXX	0	0
0057	1	00000000	0	1F	1	3E	7FFFFE	00000042	1	0	CLK	1	1FFFF	1	00420042	4	0
0058	1	00000000	0	1F	1	3E	7FFFFE	00000042	1	0	CLK	1	1FFFF	1	00420042	4	0
0059	1	00000000	0	1F	1	3E	7FFFFE	00000000	1	0	CLK	1	1FFFF	1	00420042	4	0
005A	1	D9000000	0	1F	1	3E	7FFFFE	00000000	1	0	CLK	1	1FFFF	1	D900D900	0	0
005B	1	00000000	0	1F	1	3E	7FFFFE	00000000	1	0	CLK	1	1FFFF	1	D900D900	0	0
005C	0	00000000	0	1F	1	3E	7FFFFE	00000000	1	1	CLK	1	1FFFF	1	XXXXXXXX	0	0
005D	0	00000000	0	1F	1	3E	7FFFFE	00000800	1	1	CLK	1	1FFFF	0	XXXXXXXX	0	0
005E	1	D9530000	0	1F	1	3E	7FFFFE	00000000	1	0	CLK	1	1FFFF	1	D953D953	0	0
005F	1	00000000	0	1F	1	3E	7FFFFE	00000800	1	0	CLK	1	1FFFF	1	D953D953	0	0
0060	1	00000000	0	1F	1	3E	7FFFFE	00000000	1	0	CLK	1	1FFFF	1	D953D953	0	0
0061	1	00000000	0	1F	1	3E	7FFFFE	00000000	1	0	CLK	1	1FFFF	1	D953D953	0	0
0062	1	D9530000	0	1F	1	3E	7FFFFE	00000000	1	0	CLK	1	1FFFF	1	D953D953	0	0
0063	0	00000000	0	1F	1	3E	7FFFFE	00000000	1	1	CLK	1	1FFFF	0	XXXXXXXX	0	0
0064	1	00000084	0	1F	1	3E	7FFFFE	00000000	1	0	CLK	1	1FFFF	1	08000800	8	1
0065	1	00000000	0	1F	1	3E	7FFFFE	00000800	1	0	CLK	1	1FFFF	1	08000800	8	1
0066	0	00000000	0	1F	1	3E	7FFFFE	00000000	1	1	CLK	1	1FFFF	0	XXXXXXXX	0	0
0067	1	00000000	0	1F	1	3E	7FFFFE	00000842	1	0	CLK	1	1FFFF	1	08420842	C	1
0068	1	00000000	0	1F	1	3E	7FFFFE	00000842	1	0	CLK	1	1FFFF	1	08420842	C	1
0069	1	00000000	0	1F	1	3E	7FFFFE	00000000	1	0	CLK	1	1FFFF	1	08420842	C	1
006A	0	00000000	0	1F	1	3E	7FFFFE	00000000	1	1	CLK	1	1FFFF	1	XXXXXXXX	0	0
006B	0	00000000	0	1F	1	3E	7FFFFE	00000000	1	1	CLK	1	1FFFF	1	XXXXXXXX	0	0
006C	1	00000000	0	1F	1	3E	7FFFFE	00000000	1	0	CLK	1	1FFFF	1	00840084	2	0
006D	1	00000084	0	1F	1	3E	7FFFFE	00000084	1	0	CLK	1	1FFFF	1	00840084	2	0
006E	0	00000000	0	1F	1	3E	7FFFFE	00000000	1	1	CLK	1	1FFFF	1	XXXXXXXX	0	0
006F	0	00000000	0	1F	1	3E	7FFFFE	00000000	1	1	CLK	1	1FFFF	1	XXXXXXXX	0	0
0070	0	00000000	0	1F	1	3E	7FFFFE	00000000	1	1	CLK	1	1FFFF	1	XXXXXXXX	0	0
0071	1	00000000	0	1F	1	3E	7FFFFE	00000000	1	0	CLK	1	1FFFF	1	00001000	0	0
0072	1	00000000	0	1F	1	3E	7FFFFE	00000000	1	0	CLK	1	1FFFF	1	00001000	0	0
0073	0	00000000	0	1F	1	3E	7FFFFE	00000000	1	1	CLK	1	1FFFF	1	XXXXXXXX	0	0
0074	1	00000000	0	1F	1	3E	7FFFFE	00000000	1	0	CLK	1	1FFFF	1	XXXXXXXX	0	0
0075	1	00001000	0	1F	1	3E	7FFFFE	00001000	1	0	CLK	1	1FFFF	1	00001000	0	0
0076	0	00000000	0	1F	1	3E	7FFFFE	00000000	1	1	CLK	1	1FFFF	1	XXXXXXXX	0	0
0077	0	00000000	0	1F	1	3E	7FFFFE	00000000	1	1	CLK	1	1FFFF	1	XXXXXXXX	0	0
0078	0	00000000	0	1F	1	3E	7FFFFE	00000000	1	1	CLK	1	1FFFF	1	XXXXXXXX	0	0
0079	1	00000000	0	1F	1	3E	7FFFFE	00001000	1	0	CLK	1	1FFFF	1	10001000	2	0
007A	1	00000000	0	1F	1	3E	7FFFFE	00000000	1	0	CLK	1	1FFFF	1	XXXXXXXX	0	0
007B	0	00000000	0	1F	1	3E	7FFFFE	00000000	1	1	CLK	1	1FFFF	0	XXXXXXXX	0	0
007C	0	00000000	0	3E	1	3E	7FFFFF	00000000	1	1	CLK	1	1FFFF	0	6CA96CA9	0	0

```
.. 007D 1 00001084 0 1F 1 3E 7FFFFF 00000000 0 0 1 1 1 CLK 1 1FFFF 1 XXXXXXXX 0 0 00001084 1 1 0
.. 007E 1 00000000 0 1F 1 3E 7FFFFF 00000000 0 0 1 1 1 CLK 1 1FFFF 1 XXXXXXXX 0 0 00001084 1 1 0
.. 007F 0 00000000 0 1F 1 3E 7FFFFF 00000000 1 0 1 1 0 CLK 1 1FFFF 0 XXXXXXXX 0 0 00001084 1 1 0
.. 0080 0 00000000 0 1F 1 3E 7FFFFF 00000000 1 0 1 1 0 CLK 1 1FFFF 0 XXXXXXXX 0 0 00001084 1 1 0
.. 0081 0 00000000 0 1F 1 3E 7FFFFF 00000000 1 1 1 1 0 CLK 1 1FFFF 0 XXXXXXXX 0 0 00001084 1 1 0
.. 0082 0 00000000 0 1F 1 3E 7FFFFF 00000000 0 1 1 1 0 CLK 1 1FFFF 1 XXXXXXXX 0 0 XXXXXXXX 1 1 1
.. 0083 0 00000000 0 1F 1 3E 7FFFFF 00001084 0 1 1 1 0 CLK 1 1FFFF 1 XXXXXXXX 3 1 XXXXXXXX 1 0 1
.. 0084 0 00001084 0 1F 1 3E 7FFFFF 00001084 0 1 1 1 0 CLK 1 1FFFF 1 10841084 3 1 XXXXXXXX 1 1 1
.. 0085 0 00001084 0 1F 1 3E 7FFFFF 00000000 1 1 1 1 0 CLK 1 1FFFF 0 10841084 0 0 XXXXXXXX 1 1 1
.. 0086 0 00000000 0 1F 1 3E 7FFFFF 00000000 1 1 1 1 0 CLK 1 1FFFF 0 XXXXXXXX 1 1 XXXXXXXX 1 1 0
.. 0087 0 00000000 0 1F 1 3E 7FFFFF 00000000 0 1 1 1 1 CLK 1 1FFFF 1 00000008 0 0 00000008 1 1 0
.. 0088 1 00000008 0 1F 1 3E 7FFFFF 00000000 0 1 1 1 1 CLK 1 1FFFF 1 00000008 0 0 00000008 1 1 0
.. 0089 0 00000000 0 1F 1 3E 7FFFFF 00000000 0 0 1 1 1 CLK 1 1FFFF 1 XXXXXXXX 0 0 XXXXXXXX 1 1 0
.. 008A 1 00000000 0 1F 1 3E 7FFFFF 00000008 0 0 1 1 1 CLK 1 1FFFF 1 XXXXXXXX 0 0 XXXXXXXX 1 1 0
.. 008B 1 00000000 0 1F 1 3E 7FFFFF 00000008 1 0 1 1 0 CLK 1 1FFFF 1 XXXXXXXX 1 1 XXXXXXXX 1 1 0
.. 008C 1 00000008 0 1F 1 3E 7FFFFF 00000000 1 0 1 1 0 CLK 1 1FFFF 0 00080008 1 1 XXXXXXXX 1 1 1
.. 008D 1 00000008 0 1F 1 3E 7FFFFF 00000000 1 1 1 1 0 CLK 1 1FFFF 0 00080008 0 0 XXXXXXXX 1 1 1
.. 008E 0 00000000 0 1F 1 3E 7FFFFF 00000000 0 1 1 1 0 CLK 1 1FFFF 1 XXXXXXXX 0 0 XXXXXXXX 1 1 1
.. 008F 1 00000000 0 1F 1 3E 7FFFFF 00000000 0 1 1 1 0 CLK 1 1FFFF 1 XXXXXXXX 2 1 XXXXXXXX 1 1 1
.. 0090 1 00002100 0 1F 1 3E 7FFFFF 00000000 0 1 1 1 0 CLK 1 1FFFF 1 00002100 2 1 XXXXXXXX 1 1 0
.. 0091 1 00000000 0 1F 1 3E 7FFFFF 00000000 1 1 1 1 0 CLK 1 1FFFF 0 00002100 0 0 XXXXXXXX 1 1 0
.. 0092 1 00000000 0 1F 1 3E 7FFFFF 00000000 1 1 1 1 0 CLK 1 1FFFF 0 XXXXXXXX 0 0 XXXXXXXX 1 1 0
.. 0093 0 00000000 0 1F 1 3E 7FFFFF 00000000 0 1 1 1 1 CLK 1 1FFFF 1 XXXXXXXX 0 0 XXXXXXXX 1 1 0
.. 0094 1 00002100 0 1F 1 3E 7FFFFF 00002100 0 1 1 1 1 CLK 1 1FFFF 1 21002100 2 1 XXXXXXXX 1 1 1
.. 0095 0 00002100 0 1F 1 3E 7FFFFF 00002100 0 0 1 1 1 CLK 1 1FFFF 1 21002100 0 0 XXXXXXXX 1 1 1
.. 0096 0 00000000 0 1F 1 3E 7FFFFF 00000000 0 0 1 1 1 CLK 1 1FFFF 1 XXXXXXXX 0 0 XXXXXXXX 1 1 0
.. 0097 0 00000000 0 1F 1 3E 7FFFFF 00000000 0 0 1 1 1 CLK 1 1FFFF 1 XXXXXXXX 3 1 XXXXXXXX 1 1 0
.. 0098 1 00002108 0 1F 1 3E 7FFFFF 00000000 1 0 1 1 0 CLK 1 1FFFF 1 00002108 3 1 XXXXXXXX 1 1 0
.. 0099 0 00000000 0 1F 1 3E 7FFFFF 00000000 1 0 1 1 0 CLK 1 1FFFF 0 00002108 0 0 XXXXXXXX 1 1 0
.. 009A 0 00000000 0 1F 1 3E 7FFFFF 00000000 1 1 1 1 0 CLK 1 1FFFF 0 00002108 0 0 XXXXXXXX 1 1 0
.. 009B 0 00000000 0 1F 1 3E 7FFFFF 00000000 0 1 1 1 0 CLK 1 1FFFF 1 00002108 0 0 00002108 1 1 0
.. 009C 0 00000000 0 1F 1 3E 7FFFFF 00000000 0 1 1 1 0 CLK 1 1FFFF 1 XXXXXXXX 0 0 00002108 1 1 0
.. 009D 0 00000000 0 1F 1 3E 7FFFFF 00000000 1 1 1 1 0 CLK 1 1FFFF 0 XXXXXXXX 0 0 XXXXXXXX 1 1 1
.. 009E 0 00000000 0 1F 1 3E 7FFFFF 00002108 1 1 1 1 0 CLK 1 1FFFF 0 XXXXXXXX 0 0 XXXXXXXX 1 0 1
.. 009F 1 00000000 0 1F 1 3E 7FFFFF 00002108 0 0 1 1 0 CLK 1 1FFFF 1 21082108 3 1 XXXXXXXX 1 1 1
.. 00A0 1 99530842 0 1F 1 3E 7FFFFF 00000000 1 1 1 1 0 CLK 1 1FFFF 0 21082108 0 0 XXXXXXXX 1 1 1

-- SLAVE, D32 data interface test.

.. 00A1 1 00000000 0 1F 1 3E 7FFFFE 00000000 0 0 1 0 0 CLK 1 1FFFF 0 XXXXXXXX 0 0 XXXXXXXX 1 1 1
.. 00A2 1 00000000 0 1F 1 3E 7FFFFE 00000000 0 0 1 1 1 CLK 1 1FFFF 1 XXXXXXXX F 1 XXXXXXXX 1 1 1
.. 00A3 1 4CA98421 0 1F 1 3E 7FFFFE 00000000 0 0 1 1 1 CLK 1 1FFFF 1 XXXXXXXX 0 0 4CA98421 1 1 0
.. 00A4 1 00000000 0 1F 1 3E 7FFFFE 00000000 1 0 1 1 0 CLK 1 1FFFF 1 XXXXXXXX 0 0 4CA98421 1 1 0
.. 00A5 1 00000000 0 1F 1 3E 7FFFFE 00000000 1 0 1 1 0 CLK 1 1FFFF 1 XXXXXXXX 0 0 XXXXXXXX 1 1 0
.. 00A6 1 4CA98421 0 1F 1 3E 7FFFFE 4CA98421 1 1 1 1 0 CLK 0 1FFFF 0 XXXXXXXX 0 0 XXXXXXXX 1 1 0
.. 00A7 1 4CA98421 0 1F 1 3E 7FFFFE 4CA98421 1 1 1 1 0 CLK 0 1FFFF 0 4CA98421 F 1 XXXXXXXX 1 0 1
.. 00A8 1 00000000 0 1F 1 3E 7FFFFE 00000000 0 1 1 1 0 CLK 0 1FFFF 1 4CA98421 0 0 XXXXXXXX 1 1 1
.. 00A9 1 00000000 0 1F 1 3E 7FFFFE 00000000 1 1 1 1 0 CLK 0 1FFFF 0 XXXXXXXX 0 0 XXXXXXXX 1 1 1
.. 00AA 1 00000000 0 1F 1 3E 7FFFFE 00000000 0 0 1 0 1 CLK 1 1FFFF 1 XXXXXXXX F 1 XXXXXXXX 1 1 0
.. 00AB 1 99530842 0 1F 1 3E 7FFFFE 00000000 0 0 1 0 1 CLK 1 1FFFF 1 XXXXXXXX 0 0 99530842 1 1 0
```

	Addr																				
..	00AC	1	00000000	0	1F	1	3E	7FFFFE	00000000	0	CLK	1	1FFFF	1	XXXXXXXX	0	0	99530842	1	0	0
..	00AD	1	00000000	0	1F	1	3E	7FFFFE	00000000	0	CLK	1	1FFFF	1	XXXXXXXX	0	0	XXXXXXXX	1	1	0
..	00AE	1	00000000	0	1F	1	3E	7FFFFE	99530842	1	CLK	1	1FFFF	1	XXXXXXXX	0	0	XXXXXXXX	1	1	1
..	00AF	1	00000000	0	1F	1	3E	7FFFFE	99530842	0	CLK	0	1FFFF	1	99530842	0	0	XXXXXXXX	1	0	1
..	00B0	1	00000000	0	1F	1	3E	7FFFFE	00000000	1	CLK	1	1FFFF	0	99530842	1	0	XXXXXXXX	1	1	1
..	00B1	1	00000000	0	1F	1	3E	7FFFFE	00000000	0	CLK	1	1FFFF	1	XXXXXXXX	0	0	32A61084	1	1	1
..	00B2	1	00000000	0	1F	1	3E	7FFFFE	00000000	0	CLK	1	1FFFF	1	32A61084	0	0	32A61084	1	0	1
..	00B3	1	32A61084	0	1F	1	3E	7FFFFE	32A61084	1	CLK	1	1FFFF	1	32A61084	0	F	32A61084	1	1	0
..	00B4	1	00000000	0	1F	1	3E	7FFFFE	00000000	0	CLK	1	1FFFF	1	XXXXXXXX	0	0	XXXXXXXX	1	1	0
..	00B5	1	00000000	0	1F	1	3E	7FFFFE	00000000	0	CLK	1	1FFFF	1	XXXXXXXX	0	0	XXXXXXXX	1	1	1
..	00B6	1	00000000	0	1F	1	3E	7FFFFE	00000000	0	CLK	1	1FFFF	1	XXXXXXXX	0	0	XXXXXXXX	1	0	1
..	00B7	1	00000000	0	1F	1	3E	7FFFFE	00000000	1	CLK	1	1FFFF	0	32A61084	0	0	XXXXXXXX	1	0	1
..	00B8	1	00000000	0	1F	1	3E	7FFFFE	00000000	0	CLK	1	1FFFF	1	32A61084	0	0	XXXXXXXX	1	1	0
..	00B9	1	00000000	0	1F	1	3E	7FFFFE	00000000	0	CLK	1	1FFFF	0	XXXXXXXX	F	1	XXXXXXXX	1	1	0
..	00BA	1	654C2108	0	1F	1	3E	7FFFFE	00000000	0	CLK	1	1FFFF	0	XXXXXXXX	0	0	654C2108	1	1	1
..	00BB	1	00000000	0	1F	1	3E	7FFFFE	00000000	1	CLK	1	1FFFF	0	XXXXXXXX	0	0	654C2108	1	0	1
..	00BC	1	00000000	0	1F	1	3E	7FFFFE	00000000	1	CLK	1	1FFFF	0	XXXXXXXX	0	0	654C2108	1	0	1
..	00BD	1	00000000	0	1F	1	3E	7FFFFE	00000000	0	CLK	1	1FFFF	1	654C2108	0	0	XXXXXXXX	1	1	0
..	00BE	1	00000000	0	1F	1	3E	7FFFFE	00000000	1	CLK	1	1FFFF	1	654C2108	0	F	XXXXXXXX	1	1	0
..	00BF	1	00000000	0	1F	1	3E	7FFFFE	654C2108	1	CLK	1	1FFFF	1	XXXXXXXX	0	0	XXXXXXXX	1	0	1
..	00C0	1	00000000	0	1F	1	3E	7FFFFE	00000000	0	CLK	1	1FFFF	0	654C2108	0	0	654C2108	1	1	1

```
-- File name:                -- ...........

--                    control logic.

--            ....  6 2002
--                   Peterson - 17 JAN 2004

--                  .0.0

--                Attribute(s):
--              -----------------
--                LDSIZE:32
--             .2 D16(EO) D08(O)
--          ....  AL SUPPORTED

--            .  Silicore Corporation

--              ~e software; you can
--              /or modify it under the terms
--              aneral Public License version
--                 the Free Software Foundation.

--            This library is distributed in the hope that it
--            will be useful, but WITHOUT ANY WARRANTY; without
--            even the implied warranty of MERCHANTABILITY or
--            FITNESS FOR A PARTICULAR PURPOSE.  See the GNU
--            Lesser General Public License for more details.

--            You should have received a copy of the GNU Lesser
--            General Public License along with this library;
--            if not, write to the Free Software Foundation,
--            Inc., 59 Temple Place, Suite 330, Boston, MA
--            02111-1307  USA

-- Support:   Support for this software in the form of
--            maintenance, system integration, consulting and
--            training is available for a fee from Silicore
--            Corporation.  For more information please refer
--            to the Silicore web site at: www.silicore.net.

-- Load the IEEE 1164 library and make it visible.

library ieee;
use ieee.std_logic_1164.all;
```

```vhdl
-----------------------------------------------------------------
-- Entity / port definition.
-----------------------------------------------------------------

entity BUSSCNTC is

    port(
            A:          out std_logic_vector( 1 downto 1 );
            ALOAD:      out std_logic;
            D16EO:      out std_logic;
            DLOAD:      out std_logic;
            SLAD:       in  std_logic_vector( 1 downto 1 );
            SLALD:      in  std_logic;
            SLD16EO:    in  std_logic;
            SLDLD:      in  std_logic;
            SLVLD:      in  std_logic;
            VLOAD:      out std_logic
        );

end entity BUSSCNTC;

-----------------------------------------------------------------
-- Architecture definition.
-----------------------------------------------------------------

architecture BUSSCNTC1 of BUSSCNTC is

begin

    -----------------------------------------------------------------
    -- Bus control logic.
    -----------------------------------------------------------------

    BUS_CONTROL: process( SLAD, SLALD, SLD16EO, SLDLD, SLVLD )
    begin

        ALOAD   <= SLALD;

        DLOAD   <= SLDLD;

        VLOAD   <= SLVLD;

        D16EO   <= SLD16EO;

        A(1)    <= SLAD(1);

    end process BUS_CONTROL;
```

```vhdl
end architecture BUSSCNTC1;
```

```
-- File name:        LoclAdrC.vhd
--
-- Description:       Local address bus generator.
--
-- Created on:        Thu Feb 14 10:03:06 2002
--
-- Source version:   VMEcore(tm) VER: 1.0.0
--
-- Interface:         Type:                  Attribute(s):
--                    ------                 -------------
--                    LOCAL:   LASIZE:19 LDSIZE:32
--                    SLAVE:   A24:SGL D32 D16(EO) D08(O)
--                    SLAVE:   BERR* SIGNAL SUPPORTED
--
-- License:           Copyright (C) 2002 Silicore Corporation
--
--                    This library is free software; you can
--                    redistribute it and/or modify it under the terms
--                    of the GNU Lesser General Public License version
--                    2.1 as published by the Free Software Foundation.
--
--                    This library is distributed in the hope that it
--                    will be useful, but WITHOUT ANY WARRANTY; without
--                    even the implied warranty of MERCHANTABILITY or
--                    FITNESS FOR A PARTICULAR PURPOSE.  See the GNU
--                    Lesser General Public License for more details.
--
--                    You should have received a copy of the GNU Lesser
--                    General Public License along with this library;
--                    if not, write to the Free Software Foundation,
--                    Inc., 59 Temple Place, Suite 330, Boston, MA
--                    02111-1307 USA
--
-- Support:           Support for this software in the form of
--                    maintenance, system integration, consulting and
--                    training is available for a fee from Silicore
--                    Corporation.  For more information please refer
--                    to the Silicore web site at: www.silicore.net.
--
-- Load the IEEE 1164 library and make it visible.
--
library ieee;
use ieee.std_logic_1164.all;
```

```vhdl
-- Entity / port definition.
-----------------------------------------------------------------------------

entity LOCLADRC is

    port(
            ADR_O:      out std_logic_vector( 18 downto  2 ) ;
            ALOAD:      in  std_logic;
            CLK_I:      in  std_logic;
            VA_I:       in  std_logic_vector( 18 downto  2 )
        );

end entity LOCLADRC;

-----------------------------------------------------------------------------
-- Architecture definition.
-----------------------------------------------------------------------------

architecture LOCLADRC1 of LOCLADRC is

begin

-----------------------------------------------------------------------------
-- Process for local address registers and/or counters.
-----------------------------------------------------------------------------

ADR_REG: process( CLK_I )
begin
    if( rising_edge(CLK_I) ) then if( ALOAD = '1' ) then ADR_O(  2) <= VA_I(  2); end if; end if;
    if( rising_edge(CLK_I) ) then if( ALOAD = '1' ) then ADR_O(  3) <= VA_I(  3); end if; end if;
    if( rising_edge(CLK_I) ) then if( ALOAD = '1' ) then ADR_O(  4) <= VA_I(  4); end if; end if;
    if( rising_edge(CLK_I) ) then if( ALOAD = '1' ) then ADR_O(  5) <= VA_I(  5); end if; end if;
    if( rising_edge(CLK_I) ) then if( ALOAD = '1' ) then ADR_O(  6) <= VA_I(  6); end if; end if;
    if( rising_edge(CLK_I) ) then if( ALOAD = '1' ) then ADR_O(  7) <= VA_I(  7); end if; end if;
    if( rising_edge(CLK_I) ) then if( ALOAD = '1' ) then ADR_O(  8) <= VA_I(  8); end if; end if;
    if( rising_edge(CLK_I) ) then if( ALOAD = '1' ) then ADR_O(  9) <= VA_I(  9); end if; end if;
    if( rising_edge(CLK_I) ) then if( ALOAD = '1' ) then ADR_O(10) <= VA_I(10); end if; end if;
    if( rising_edge(CLK_I) ) then if( ALOAD = '1' ) then ADR_O(11) <= VA_I(11); end if; end if;
    if( rising_edge(CLK_I) ) then if( ALOAD = '1' ) then ADR_O(12) <= VA_I(12); end if; end if;
    if( rising_edge(CLK_I) ) then if( ALOAD = '1' ) then ADR_O(13) <= VA_I(13); end if; end if;
    if( rising_edge(CLK_I) ) then if( ALOAD = '1' ) then ADR_O(14) <= VA_I(14); end if; end if;
    if( rising_edge(CLK_I) ) then if( ALOAD = '1' ) then ADR_O(15) <= VA_I(15); end if; end if;
    if( rising_edge(CLK_I) ) then if( ALOAD = '1' ) then ADR_O(16) <= VA_I(16); end if; end if;
    if( rising_edge(CLK_I) ) then if( ALOAD = '1' ) then ADR_O(17) <= VA_I(17); end if; end if;
    if( rising_edge(CLK_I) ) then if( ALOAD = '1' ) then ADR_O(18) <= VA_I(18); end if; end if;

end process ADR_REG;
```

end architecture LOCLADRC1;

```
--------------------------------------------------------------------
-- File name:       LoclDatC.vhd
--
-- Description:     Data multiplexor and/or register for DAT_O().
--
-- Created on:      Thu Feb 14 10:03:06 2002
--
-- Source version:  VMEcore(tm) VER: 1.0.0
--
-- Interface:       Type:        Attribute(s):
--                  -----        -------------
--                  LOCAL:    LASIZE:19 LDSIZE:32
--                  SLAVE:    A24:SGL D32 D16(EO) D08(O)
--                  SLAVE:    BERR* SIGNAL SUPPORTED
--
-- License:         Copyright (C) 2002 Silicore Corporation
--
--                  This library is free software; you can
--                  redistribute it and/or modify it under the terms
--                  of the GNU Lesser General Public License version
--                  2.1 as published by the Free Software Foundation.
--
--                  This library is distributed in the hope that it
--                  will be useful, but WITHOUT ANY WARRANTY; without
--                  even the implied warranty of MERCHANTABILITY or
--                  FITNESS FOR A PARTICULAR PURPOSE.  See the GNU
--                  Lesser General Public License for more details.
--
--                  You should have received a copy of the GNU Lesser
--                  General Public License along with this library;
--                  if not, write to the Free Software Foundation,
--                  Inc., 59 Temple Place, Suite 330, Boston, MA
--                  02111-1307 USA
--
-- Support:         Support for this software in the form of
--                  maintenance, system integration, consulting and
--                  training is available for a fee from Silicore
--                  Corporation.  For more information please refer
--                  to the Silicore web site at: www.silicore.net.
--------------------------------------------------------------------


--------------------------------------------------------------------
-- Load the IEEE 1164 library and make it visible.
--------------------------------------------------------------------
library ieee;
use ieee.std_logic_1164.all;
--------------------------------------------------------------------
```

```vhdl
-- -----------------------------------------------------------------------
-- Entity / port definition.
-- -----------------------------------------------------------------------

entity LOCLDATC is

    port(
        CLK_I:      in  std_logic;
        D16EO:      in  std_logic;
        DAT_O:      out std_logic_vector( 31 downto  0 );
        DLOAD:      in  std_logic;
        VD_I:       in  std_logic_vector( 31 downto  0 )
    );

end entity LOCLDATC;

-- -----------------------------------------------------------------------
-- Architecture definition.
-- -----------------------------------------------------------------------

architecture LOCLDATC1 of LOCLDATC is

-- -----------------------------------------------------------------------
-- Signal definition(s).
-- -----------------------------------------------------------------------

    signal  EN1:        std_logic;

begin

-- -----------------------------------------------------------------------
-- WORD1: DAT_O( 31 downto 16 ).
-- -----------------------------------------------------------------------

WORD1: process ( CLK_I, D16EO, EN1 )
begin
    if( rising_edge(CLK_I) ) then
        if( DLOAD = '1' ) then

            EN1 <= D16EO;

            case EN1 is
                when       '0'  => DAT_O( 31 downto 16 ) <= VD_I( 31 downto 16 );
                when others     => DAT_O( 31 downto 16 ) <= VD_I( 15 downto  0 );
            end case;

        end if;
    end if;
```

```vhdl
end process WORD1;

---------------------------------------------------------
-- WORD0: DAT_O( 15 downto   0 ).
---------------------------------------------------------

WORD0: process( CLK_I )
begin

   if( rising_edge(CLK_I) ) then
      if( DLOAD = '1' ) then
         DAT_O( 15 downto   0 ) <= VD_I( 15 downto   0 );
      end if;
   end if;

end process WORD0;


end architecture LOCLDATC1;
```

```vhdl
-- File name:      SlavCntC.vhd
--
-- Description:    SLAVE SlavCnt logic.
--
-- Created on:     Thu Feb 14 10:03:06 2002
--
-- Modified:       Hand modified to support a data direction (DDIR)
--                 signal on March 1, 2002 - WD Peterson.
--
-- Source version: VMEcore(tm) VER: 1.0.0
--
-- Interface:      Type:              Attribute(s):
--                 -----              -------------
--                 LOCAL:  LASIZE:19 LDSIZE:32
--                 SLAVE:  A24:SGL D32 D16(EO) D08(O)
--                 SLAVE:  BERR* SIGNAL SUPPORTED
--
-- License:        Copyright (C) 2002 Silicore Corporation
--
--                 This library is free software; you can
--                 redistribute it and/or modify it under the terms
--                 of the GNU Lesser General Public License version
--                 2.1 as published by the Free Software Foundation.
--
--                 This library is distributed in the hope that it
--                 will be useful, but WITHOUT ANY WARRANTY; without
--                 even the implied warranty of MERCHANTABILITY or
--                 FITNESS FOR A PARTICULAR PURPOSE.   See the GNU
--                 Lesser General Public License for more details.
--
--                 You should have received a copy of the GNU Lesser
--                 General Public License along with this library;
--                 if not, write to the Free Software Foundation,
--                 Inc., 59 Temple Place, Suite 330, Boston, MA
--                 02111-1307 USA
--
-- Support:        Support for this software in the form of
--                 maintenance, system integration, consulting and
--                 training is available for a fee from Silicore
--                 Corporation.   For more information please refer
--                 to the Silicore web site at: www.silicore.net.
--
---------------------------------------------------------------------
--
-- Load the IEEE 1164 library and make it visible.
--
---------------------------------------------------------------------
library ieee;
use ieee.std_logic_1164.all;
```

```
------------------------------
-- Entity / port definition.
------------------------------

entity SLAVCNTC is

  port(

    ACK_I:      in  std_logic;
    ADC:        in  std_logic;
    CLK_I:      in  std_logic;
    CYC_O:      out std_logic;
    DDIR:       out std_logic;
    ERR_I:      in  std_logic;
    RST_I:      in  std_logic;
    SEL_O:      out std_logic_vector( 3 downto 0 );
    SLAD:       out std_logic_vector( 1 downto 1 );
    SIALD:      out std_logic;
    SLD16EO:    out std_logic;
    SLDLD:      out std_logic;
    SLVLD:      out std_logic;
    STB_O:      out std_logic;
    VA_I:       in  std_logic_vector( 1 downto 1 );
    VNAS_I:     in  std_logic;
    VNBERR_O:   out std_logic;
    VNDS0_I:    in  std_logic;
    VNDS1_I:    in  std_logic;
    VNDTACK_O:  out std_logic;
    VNLWORD_I:  in  std_logic;
    VNWRITE_I:  in  std_logic;
    WE_O:       out std_logic

    );

end entity SLAVCNTC;

------------------------------
-- Architecture definition.
------------------------------

architecture SLAVCNTC1 of SLAVCNTC is

------------------------------
-- Signal definition(s).
------------------------------

  signal A:       std_logic_vector( 1 downto 1 );
  signal ACK:     std_logic;
  signal ALOAD:   std_logic;
```

```vhdl
signal AQ:      std_logic_vector( 1 downto 1 );
signal AS:      std_logic;
signal BERR:    std_logic;
signal BMC:     std_logic;
signal CYC:     std_logic;
signal DIR:     std_logic;
signal D16EO:   std_logic;
signal D32:     std_logic;
signal DLOAD:   std_logic;
signal DREAD:   std_logic;
signal DS:      std_logic;
signal DTACK:   std_logic;
signal DTKCD:   std_logic;
signal IHLD:    std_logic;
signal MUX:     std_logic;
signal NAS:     std_logic;
signal NDS0:    std_logic;
signal NDS1:    std_logic;
signal NLWORD:  std_logic;
signal P:       std_logic_vector( 2 downto 0 );
signal SEL:     std_logic_vector( 3 downto 0 );
signal SELD:    std_logic;
signal STB:     std_logic;

begin

--------------------------------------------------------
-- Common logic used by other processes.
--------------------------------------------------------

TERMINATOR: process ( ACK_I, ERR_I )
begin

    ACK  <= ACK_I or ERR_I;

end process TERMINATOR;

--------------------------------------------------------
-- Synchronizer for address and data strobes.
--------------------------------------------------------

SYNC: process ( CLK_I, NAS, NDS0, NDS1 )
begin

    if( rising_edge(CLK_I) ) then NAS  <= VNAS_I;  end if;
    if( rising_edge(CLK_I) ) then NDS0 <= VNDS0_I; end if;
    if( rising_edge(CLK_I) ) then NDS1 <= VNDS1_I; end if;
```

```vhdl
        AS    <= not(NAS);
        DS    <= not(NDS0) or not(NDS1);

    end process SYNC;

    ---------------------------------------
    -- Sequencer.
    ---------------------------------------

    SEQU: process( ALOAD, CLK_I, DLOAD, DS, MUX, P )
    begin

        ---------------------------------------
        -- Internal signals.
        ---------------------------------------

        IHLD <= '0';
        BMC  <= '0';
        MUX  <= '0';

        ---------------------------------------
        -- State machine.
        ---------------------------------------

        if( rising_edge(CLK_I) ) then

            P(0) <= ( not(P(2))                                             and AS      and DS      )
                 or (     P(2)  and not(P(1))                               and AS      and DS      );

            P(1) <= ( not(P(2)) and     P(1)                                and         and DS      )
                 or ( not(P(2)) and     P(1)  and not(P(0))                 and AS      and not(DS) )
                 or ( not(P(2)) and not(P(1)) and     P(0)                  and AS      and not(DS) )
                 or (                not(P(1)) and     P(0)                 and not(AS) and DS      )
                 or (     P(2)  and     P(1)                                and         and DS      )
                 or (     P(2)  and     P(1)  and     P(0)                  and not(AS) and DS      )
                 or ( not(P(2)) and     P(1)  and not(P(0)) and IHLD        and AS      and DS      );

            P(2) <= ( not(P(2)) and     P(1)  and not(P(1))                 and         and DS      )
                 or ( not(P(2)) and            P(1)                         and not(AS) and DS      )
                 or ( not(P(2)) and            P(2)                         and not(AS) and DS      )
                 or (     P(2)                                              and         and DS      )
                 or ( not(P(2)) and     P(1)  and not(P(0))                 and not(AS) and DS      )
                 or (     P(2)  and     P(1)  and     P(0)                  and AS      and DS      )
                 or (     P(2)  and            and not(P(1)) and IHLD       and AS      and not(DS) )
                 or ( not(P(2)) and     P(1)  and not(P(0)) and IHLD        and AS      and DS      );
```

```vhdl
    end if;

    --
    ---------------------------------------------------------------
    -- Control signals.
    ---------------------------------------------------------------
    --
    ALOAD <= ( not(P(2)) and not(P(1))    and  DS  and not(IHLD) );

    SLALD <= ALOAD;

    DLOAD <= ( not(P(2)) and not(P(1)) and not(P(0)) and not(MUX) and DS and not(IHLD) )
    or ( not(P(2)) and     P(1)  and not(P(0))                   and DS and not(IHLD) )
    or (     P(2)  and not(P(1)) and not(P(0))                   and DS and not(IHLD) );

    SLDLD <= DLOAD;

end process SEQU;

--
---------------------------------------------------------------
-- Data cycle function generator.
---------------------------------------------------------------
--
DCYCL: process ( A, D16EO, NLWORD, VNDS0_I, VNDS1_I )
begin

    D32    <= not(VNDS1_I) and not(VNDS0_I) and not(NLWORD) and not(A(1)) ;

    D16EO   <= ( not(VNDS1_I) and NLWORD )
    or ( not(VNDS0_I) and NLWORD ) ;

    SLD16EO <= D16EO;

end process DCYCL;

--
---------------------------------------------------------------
-- Select signal(s).
---------------------------------------------------------------
--
SEL_GENR: process ( CLK_I, SEL )
begin

    if( rising_edge(CLK_I) ) then

        SEL(3) <= ( DLOAD and ADC and D32 )
        or ( DLOAD and ADC and D16EO and not(VNDS1_I) and not(A(1)) )
        or ( SEL(3) and DS and not(ACK) and not(RST_I) ) ;
```

```vhdl
    end if;

    SEL_O(3) <= SEL(3);

    if( rising_edge(CLK_I) ) then

        SEL(2) <= ( DLOAD and ADC and D32 )
               or ( DLOAD and ADC and D16EO  and not(VNDS0_I) and not(A(1)) )
               or ( SEL(2) and DS and not(ACK) and not(RST_I) );

    end if;

    SEL_O(2) <= SEL(2);

    if( rising_edge(CLK_I) ) then

        SEL(1) <= ( DLOAD and ADC and D32 )
               or ( DLOAD and ADC and D16EO and not(VNDS1_I) and A(1) )
               or ( SEL(1) and DS and not(ACK) and not(RST_I) );

    end if;

    SEL_O(1) <= SEL(1);

    if( rising_edge(CLK_I) ) then

        SEL(0) <= ( DLOAD and ADC and D32 )
               or ( DLOAD and ADC and D16EO and not(VNDS0_I) and A(1) )
               or ( SEL(0) and DS and not(ACK) and not(RST_I) );

    end if;

    SEL_O(0) <= SEL(0);

end process SEL_GENR;

-----------------------------------------------------------------
-- Strobe(s).
-----------------------------------------------------------------
STROBES: process( ACK, AS, CLK_I, CYC, DS, STB )
begin
-----------------------------------------------------------------
```

```vhdl
    -- CYC_O.
    --------------------------------------------------------------------

    if( rising_edge(CLK_I) ) then

        CYC_O <= (ALOAD and (ADC)) or (CYC and AS and not(RST_I));

    end if;

    CYC_O <= CYC and AS;

    --------------------------------------------------------------------
    -- STB_O.
    --------------------------------------------------------------------

    if( rising_edge(CLK_I) ) then

        STB_O <= (DLOAD and (ADC)) or (STB and DS and not(ACK) and not(RST_I));

    end if;

    STB_O <= STB and DS;

    --------------------------------------------------------------------
    -- WE_O.
    --------------------------------------------------------------------

    if( rising_edge(CLK_I) ) then if(DLOAD = '1') then

        WE_O <= not(VNWRITE_I);

    end if; end if;

end process STROBES;

    --------------------------------------------------------------------
    -- Data direction signal (DDIR).    This signal was added by hand
    -- on March 1, 2002 in order to support a data direction line
    -- on external buffer chips (for the VMEbus data lines).
    --------------------------------------------------------------------

DDIR_SIGNAL: process( CLK_I, DIR, DS )
begin

    if( rising_edge( CLK_I ) ) then
```

```vhdl
                DIR <=  (  STB and VNWRITE_I and DS )
                    or (  DIR                and DS );

            end if;

            DDIR <= DIR and DS;

        end process DDIR_SIGNAL;

--------------------------------------------------------------------
-- DTACK generator.
--------------------------------------------------------------------

DTACK_GENR: process( ACK_I, ADC, CLK_I, DS, DTACK, DTKCD, STB, VNWRITE_I )
begin

        SELD <= ADC and DS;

        DTKCD <= VNWRITE_I;

        if( rising_edge(CLK_I) ) then

            DREAD <= ( not(RST_I) and not(DTACK)  and not(DREAD)  and ACK_I and           DTKCD  and DS and STB and SELD );

            DTACK <= ( not(RST_I) and not(DTACK)  and not(DREAD)  and ACK_I and not(DTKCD)  and DS and SELD )
                  or ( not(RST_I) and not(DTACK)  and           DREAD                              and DS and SELD )
                  or ( not(RST_I) and           DTACK  and not(DREAD)                              and DS and SELD );

            BERR  <= ( not(RST_I) and not(BERR)   and ERR_I and DS )
                  or ( not(RST_I) and           BERR                and DS );

        end if;

        VNDTACK_O <= not(DTACK and DS);
        VNBERR_O  <= not(BERR and DS);

        SLVLD <= STB and ACK_I;

    end process DTACK_GENR;

--------------------------------------------------------------------
-- Miscellaneous address control signals.
--------------------------------------------------------------------

misc_addr: process( A, ALOAD, AQ, VA_I, VNLWORD_I )
begin

        if ALOAD = '1' then NLWORD <= VNLWORD_I; end if;
```

```vhdl
      if ALOAD = '1' then AQ(1) <= VA_I(1); end if;

      A(1) <= AQ(1);

      SLAD(1) <= A(1);

   end process misc_addr;

end architecture SLAVCNTCl;
```

```
-- File name:        SlavDecC.vhd
--
-- Description:      Slave address decode logic.
--
-- Created on:       Thu Feb 14 10:03:06 2002
--
-- Source version:   VMEcore(tm) VER: 1.0.0
--
-- Interface:        Type:            Attribute(s):
--                   -----            -------------
--                   LOCAL:   LASIZE:19 LDSIZE:32
--                   SLAVE:   A24:SGL D32 D16(EO) D08(O)
--                   SLAVE:   BERR* SIGNAL SUPPORTED
--
-- License:          Copyright (C) 2002 Silicore Corporation
--
--                   This library is free software; you can
--                   redistribute it and/or modify it under the terms
--                   of the GNU Lesser General Public License version
--                   2.1 as published by the Free Software Foundation.
--
--                   This library is distributed in the hope that it
--                   will be useful, but WITHOUT ANY WARRANTY; without
--                   even the implied warranty of MERCHANTABILITY or
--                   FITNESS FOR A PARTICULAR PURPOSE.   See the GNU
--                   Lesser General Public License for more details.
--
--                   You should have received a copy of the GNU Lesser
--                   General Public License along with this library;
--                   if not, write to the Free Software Foundation,
--                   Inc., 59 Temple Place, Suite 330, Boston, MA
--                   02111-1307 USA
--
-- Support:          Support for this software in the form of
--                   maintenance, system integration, consulting and
--                   training is available for a fee from Silicore
--                   Corporation.  For more information please refer
--                   to the Silicore web site at: www.silicore.net.
------------------------------------------------------------------

------------------------------------------------------------------
-- Load the IEEE 1164 library and make it visible.
------------------------------------------------------------------
library ieee;
use ieee.std_logic_1164.all;
------------------------------------------------------------------
```

```vhdl
-- -----------------------------------------------------------------------------
-- Entity / port definition.
-- -----------------------------------------------------------------------------

entity SLAVDECC is

    port(
        A24SGL_O:       out std_logic;
        ADC:            out std_logic;
        ALOAD:          in  std_logic;
        CLK_I:          in  std_logic;
        SAC24_I:        in  std_logic;
        SAE24_I:        in  std_logic;
        VAM_I:          in  std_logic_vector( 23 downto 19 );
        VA_I:           in  std_logic_vector(  5 downto  0 );
        VNIACK_I:       in  std_logic_vector( 23 downto 19 );
        );

end entity SLAVDECC;

-- -----------------------------------------------------------------------------
-- Architecture definition.
-- -----------------------------------------------------------------------------

architecture SLAVDECC1 of SLAVDECC is

-- -----------------------------------------------------------------------------
-- Signal definition(s).
-- -----------------------------------------------------------------------------

    signal A24SGL:      std_logic;
    signal AD:          std_logic;
    signal SCMP24:      std_logic;

begin

    AM_CODES: process( CLK_I, VAM_I, VNIACK_I )
    begin

    -- -----------------------------------------------------------------------------
    -- Address modifier code, decode logic.
    -- -----------------------------------------------------------------------------

    A24SGL <=      ( VNIACK_I and     VAM_I(5) and     VAM_I(4) and     VAM_I(3) and     VAM_I(2) and     VAM_I(1)
    and not(VAM_I(0)) ) (     VNIACK_I and     VAM_I(5) and     VAM_I(4) and     VAM_I(3) and     VAM_I(2) and not(VAM_I(1))
    and       VAM_I(0)    ) (     VNIACK_I and     VAM_I(5) and     VAM_I(4) and     VAM_I(3) and not(VAM_I(2)) and     VAM_I(1)
    and not(VAM_I(0)) )      or ( VNIACK_I and     VAM_I(5) and     VAM_I(4) and     VAM_I(3) and not(VAM_I(2)) and     VAM_I(1)
```

```vhdl
and         or ( VNIACK_I and    VAM_I(5)    and    VAM_I(4)    and    VAM_I(3)    and not(VAM_I(2))    and not(VAM_I(1))
            VAM_I(0)  );

        if( rising_edge(CLK_I) ) then if( ALOAD = '1' ) then A24SGL_O <= A24SGL; end if; end if;

    end process AM_CODES;

    -------------------------------
    -- SLAVE A24 address comparator.
    -------------------------------

    SLAVE_A24CMP: process( SAC24_I, SAE24_I, VA_I )
    begin

        SCMP24 <=   ( SAC24_I(23) xnor VA_I(23) )
                and ( SAC24_I(22) xnor VA_I(22) )
                and ( SAC24_I(21) xnor VA_I(21) )
                and ( SAC24_I(20) xnor VA_I(20) )
                and ( SAC24_I(19) xnor VA_I(19) )
                and   SAE24_I;

    end process SLAVE_A24CMP;

    -------------------------------
    -- ADC decoder.
    -------------------------------

    ADC_DEC: process( A24SGL, AD, SCMP24 )
    begin

        AD  <= ( (A24SGL) and SCMP24 );

        ADC <= AD;

    end process ADC_DEC;

end architecture SLAVDECC1;
```

```vhdl
-- File name:        VmebADbC.vhd
--
-- Description:      Data multiplexor and/or register for VD_O().
--
-- Created on:       Thu Feb 14 10:03:06 2002
--
-- Source version:   VMEcore(tm) VER: 1.0.0
--
-- Interface:        Type:               Attribute(s):
--                   ----                -------------
--                   LOCAL:   LASIZE:19 LDSIZE:32
--                   SLAVE:   A24:SGL D32 D16(EO) D08(O)
--                   SLAVE:   BERR* SIGNAL SUPPORTED
--
-- License:          Copyright (C) 2002 Silicore Corporation
--
--                   This library is free software; you can
--                   redistribute it and/or modify it under the terms
--                   of the GNU Lesser General Public License version
--                   2.1 as published by the Free Software Foundation.
--
--                   This library is distributed in the hope that it
--                   will be useful, but WITHOUT ANY WARRANTY; without
--                   even the implied warranty of MERCHANTABILITY or
--                   FITNESS FOR A PARTICULAR PURPOSE.  See the GNU
--                   Lesser General Public License for more details.
--
--                   You should have received a copy of the GNU Lesser
--                   General Public License along with this library;
--                   if not, write to the Free Software Foundation,
--                   Inc., 59 Temple Place, Suite 330, Boston, MA
--                   02111-1307 USA
--
-- Support:          Support for this software in the form of
--                   maintenance, system integration, consulting and
--                   training is available for a fee from Silicore
--                   Corporation.  For more information please refer
--                   to the Silicore web site at: www.silicore.net.
--
--
-- Load the IEEE 1164 library and make it visible.
--
library ieee;
use ieee.std_logic_1164.all;
```

```vhdl
-- Entity / port definition.
-----------------------------------------------------

entity VMEBADBC is

    port(

        A:          in  std_logic_vector( 1 downto 1 );
        CLK_I:      in  std_logic;
        D16EO:      in  std_logic;
        DAT_I:      in  std_logic_vector( 31 downto 0 );
        VD_O:       out std_logic_vector( 31 downto 0 );
        VLOAD:      in  std_logic
        );

end entity VMEBADBC;

-----------------------------------------------------
-- Architecture definition.
-----------------------------------------------------

architecture VMEBADBC1 of VMEBADBC is

-----------------------------------------------------
-- Signal definition(s).
-----------------------------------------------------

    signal  DLO:    std_logic;
    signal  DME:    std_logic;

begin

-----------------------------------------------------
-- VD_O( 31 downto 16 ).
-----------------------------------------------------

VDATO32: process( CLK_I )
begin

    if( rising_edge(CLK_I) ) then
        if( VLOAD = '1' ) then
            VD_O( 31 downto 16 ) <= DAT_I( 31 downto 16 );
        end if;
    end if;

end process VDATO32;

-----------------------------------------------------
```

```vhdl
--  VD_O( 15 downto  8 ).
-------------------------------------------------------------------------

VDATO16: process ( A, CLK_I, D16EO, DME )
begin

    DME  <=  (D16EO and not(A(1)))  ;

    if ( rising_edge(CLK_I) ) then
        if ( VLOAD = '1' ) then

            case DME is
                when        '0'  =>  VD_O( 15 downto  8 )  <=  DAT_I( 15 downto  8 )  ;
                when others     =>  VD_O( 15 downto  8 )  <=  DAT_I( 31 downto 24 )  ;
            end case;

        end if;
    end if;

end process VDATO16;

------------------------------
--  VD_O( 7 downto   0 ).
------------------------------

VDATO008: process ( A, CLK_I, D16EO, DLO )
begin

    DLO  <=  (D16EO and not(A(1)))  ;

    if ( rising_edge(CLK_I) ) then
        if ( VLOAD = '1' ) then

            case DLO is
                when        '0'  =>  VD_O( 7 downto  0 )  <=  DAT_I( 7 downto  0 )  ;
                when others     =>  VD_O( 7 downto  0 )  <=  DAT_I( 23 downto 16 )  ;
            end case;

        end if;
    end if;

end process VDATO008;

end architecture VMEBADBC1;
```

```vhdl
--------------------------------------------------------------------------
-- File name:       VmeCoreC.vhd
--
-- Description:     VMEcore high-level entity/architecture.
--
-- Created on:      Thu Feb 14 10:03:06 2002
--
-- Modified:        File modified by hand on MAR 01, 2002 to support
--                  a data direction (DDIR) signal - WD Peterson.
--
-- Source version:  VMEcore(tm) VER: 1.0.0
--
-- Interface:       Type:               Attribute(s):
--                  ----                -------------
--                  LOCAL:   LASIZE:19 LDSIZE:32
--                  SLAVE:   A24:SGL D32 D16(EO) D08(O)
--                  SLAVE:   BERR* SIGNAL SUPPORTED
--
-- License:         Copyright (C) 2002 Silicore Corporation
--
--                  This library is free software; you can
--                  redistribute it and/or modify it under the terms
--                  of the GNU Lesser General Public License version
--                  2.1 as published by the Free Software Foundation.
--
--                  This library is distributed in the hope that it
--                  will be useful, but WITHOUT ANY WARRANTY; without
--                  even the implied warranty of MERCHANTABILITY or
--                  FITNESS FOR A PARTICULAR PURPOSE.  See the GNU
--                  Lesser General Public License for more details.
--
--                  You should have received a copy of the GNU Lesser
--                  General Public License along with this library;
--                  if not, write to the Free Software Foundation,
--                  Inc., 59 Temple Place, Suite 330, Boston, MA
--                  02111-1307  USA
--
-- Support:         Support for this software in the form of
--                  maintenance, system integration, consulting and
--                  training is available for a fee from Silicore
--                  Corporation.  For more information please refer
--                  to the Silicore web site at: www.silicore.net.
--------------------------------------------------------------------------

--------------------------------------------------------------------------
-- Load the IEEE 1164 library and make it visible.
--------------------------------------------------------------------------
library ieee;
use ieee.std_logic_1164.all;
```

```vhdl
---------------------------------------------------------------------
-- Entity / port definition.
---------------------------------------------------------------------

entity VMECOREC is
  port(

    -- Miscellaneous signals:

    DDIR:              out std_logic;

    -- WISHBONE MASTER signals:

    A24SGL_O:          out std_logic;
    ACK_I:             in  std_logic;
    ADR_O:             out std_logic_vector( 18 downto  2 );
    CLK_I:             in  std_logic;
    CYC_O:             out std_logic;
    DAT_I:             in  std_logic_vector( 31 downto  0 );
    DAT_O:             out std_logic_vector( 31 downto  0 );
    ERR_I:             in  std_logic;
    RST_I:             in  std_logic;
    SAC24_I:           in  std_logic_vector( 23 downto 19 );
    SAE24_I:           in  std_logic;
    SEL_O:             out std_logic_vector(  3 downto  0 );
    STB_O:             out std_logic;
    WE_O:              out std_logic;

    -- VMEbus interface signals:

    VAM_I:             in  std_logic_vector(  5 downto  0 );
    VA_I:              in  std_logic_vector( 23 downto  1 );
    VD_I:              in  std_logic_vector( 31 downto  0 );
    VD_O:              out std_logic_vector( 31 downto  0 );
    VNAS_I:            in  std_logic;
    VNBERR_O:          out std_logic;
    VNDS0_I:           in  std_logic;
    VNDS1_I:           in  std_logic;
    VNDTACK_O:         out std_logic;
    VNIACK_I:          in  std_logic;
    VNLWORD_I:         in  std_logic;
    VNWRITE_I:         in  std_logic

  );
end entity VMECOREC;
---------------------------------------------------------------------
```

```
-- Architecture definition.
--------------------------------------------------------------------------------

architecture VMECOREC1 of VMECOREC is

--------------------------------------------------------------------------------
-- Component definition(s).
--------------------------------------------------------------------------------

component BUSSCNTC
port(

    A:          out std_logic_vector( 1 downto 1 );
    ALOAD:      out std_logic;
    D16EO:      out std_logic;
    DLOAD:      out std_logic;
    SLAD:       in  std_logic_vector( 1 downto 1 );
    SLALD:      in  std_logic;
    SLD16EO:    in  std_logic;
    SLDLD:      in  std_logic;
    SLVLD:      in  std_logic;
    VLOAD:      out std_logic

    );
end component BUSSCNTC;

component LOCLADRC
port(

    ADR_O:      out std_logic_vector( 18 downto 2 );
    ALOAD:      in  std_logic;
    CLK_I:      in  std_logic;
    VA_I:       in  std_logic_vector( 18 downto 2 )

    );
end component LOCLADRC;

component LOCLDATC
port(

    CLK_I:      in  std_logic;
    D16EO:      in  std_logic;
    DAT_O:      out std_logic_vector( 31 downto 0 );
    DLOAD:      in  std_logic;
    VD_I:       in  std_logic_vector( 31 downto 0 )

    );
end component LOCLDATC;

component SLAVCNTC
port(

    ACK_I:      in  std_logic;
    ADC:        in  std_logic;
    CLK_I:      in  std_logic;
    CYC_O:      out std_logic;
```

```vhdl
            DDIR:        out std_logic;
            ERR_I:       in  std_logic;
            RST_I:       in  std_logic;
            SEL_O:       out std_logic;
            SLAD:        out std_logic_vector( 3 downto 0 );
            SLALD:       out std_logic_vector( 1 downto 0 );
            SLD16EO:     out std_logic;
            SLDLD:       out std_logic;
            SLVLD:       out std_logic;
            STB_O:       out std_logic;
            VA_I:        in  std_logic_vector( 1 downto 1 );
            VNAS_I:      in  std_logic;
            VNBERR_O:    out std_logic;
            VNDS0_I:     in  std_logic;
            VNDS1_I:     in  std_logic;
            VNDTACK_O:   out std_logic;
            VNLWORD_I:   in  std_logic;
            VNWRITE_I:   in  std_logic;
            WE_O:        out std_logic
    );
end component SLAVCNTC;

component SLAVDECC
port(
            A24SGL_O:    out std_logic;
            ADC:         out std_logic;
            ALOAD:       in  std_logic;
            CLK_I:       in  std_logic;
            SAC24_I:     in  std_logic_vector( 23 downto 19 );
            SAE24_I:     in  std_logic_vector( 5 downto 0 );
            VAM_I:       in  std_logic_vector( 5 downto 0 );
            VA_I:        in  std_logic_vector( 23 downto 19 );
            VNIACK_I:    in  std_logic
    );
end component SLAVDECC;

component VMEBADBC
port(
            A:           in  std_logic_vector( 1 downto 1 );
            CLK_I:       in  std_logic;
            D16EO:       in  std_logic;
            DAT_I:       in  std_logic_vector( 31 downto 0 );
            VD_O:        out std_logic_vector( 31 downto 0 );
            VLOAD:       in  std_logic
    );
end component VMEBADBC;

-- ------------------------------------------
-- Signal definition(s).
```

```vhdl
--------------------------------------------------------------------------------
signal  A:          std_logic_vector(  1 downto  1 );
signal  ADC:        std_logic;
signal  ALOAD:      std_logic;
signal  D16EO:      std_logic;
signal  DLOAD:      std_logic;
signal  SLAD:       std_logic_vector(  1 downto  1 );
signal  SLALD:      std_logic;
signal  SLD16EO:    std_logic;
signal  SLDLD:      std_logic;
signal  SLVLD:      std_logic;
signal  VLOAD:      std_logic;

begin

--------------------------------------------------------------------------------
-- Component map.
--------------------------------------------------------------------------------
U1: component BUSSCNTC
port map(
            A           => A(  1 downto  1 ),
            ALOAD       => ALOAD,
            D16EO       => D16EO,
            DLOAD       => DLOAD,
            SLAD        => SLAD(  1 downto  1 ),
            SLALD       => SLALD,
            SLD16EO     => SLD16EO,
            SLDLD       => SLDLD,
            SLVLD       => SLVLD,
            VLOAD       => VLOAD
        );

U2: component LOCLADRC
port map(
            ADR_O       => ADR_O(  18 downto  2 ),
            ALOAD       => ALOAD,
            CLK_I       => CLK_I,
            VA_I        => VA_I(  18 downto  2 )
        );

U3: component LOCLDATC
port map(
            CLK_I       => CLK_I,
            D16EO       => D16EO,
            DAT_O       => DAT_O(  31 downto  0 ),
            DLOAD       => DLOAD,
            VD_I        => VD_I(  31 downto  0 )
```

```
        );

U4: component SLAVCNTC
port map(
        ACK_I       =>  ACK_I,
        ADC         =>  ADC,
        CLK_I       =>  CLK_I,
        CYC_O       =>  CYC_O,
        DDIR        =>  DDIR,
        ERR_I       =>  ERR_I,
        RST_I       =>  RST_I,
        SEL_O       =>  SEL_O( 3 downto 0 ),
        SIAD        =>  SIAD( 1 downto 1 ),
        SIALD       =>  SIALD,
        SLD16EO     =>  SLD16EO,
        SLDLD       =>  SLDLD,
        SLVLD       =>  SLVLD,
        STB_O       =>  STB_O,
        VA_I        =>  VA_I( 1 downto 1 ),
        VNAS_I      =>  VNAS_I,
        VNBERR_O    =>  VNBERR_O,
        VNDS0_I     =>  VNDS0_I,
        VNDS1_I     =>  VNDS1_I,
        VNDTACK_O   =>  VNDTACK_O,
        VNLWORD_I   =>  VNLWORD_I,
        VNWRITE_I   =>  VNWRITE_I,
        WE_O        =>  WE_O
        );

U5: component SLAVDECC
port map(
        A24SGL_O    =>  A24SGL_O,
        ADC         =>  ADC,
        ALOAD       =>  ALOAD,
        CLK_I       =>  CLK_I,
        SAC24_I     =>  SAC24_I( 23 downto 19 ),
        SAE24_I     =>  SAE24_I,
        VAM_I       =>  VAM_I( 5 downto 0 ),
        VA_I        =>  VA_I( 23 downto 19 ),
        VNIACK_I    =>  VNIACK_I
        );

U6: component VMEBADBC
port map(
        A           =>  A( 1 downto 1 ),
        CLK_I       =>  CLK_I,
        D16EO       =>  D16EO,
        DAT_I       =>  DAT_I( 31 downto 0 ),
        VD_O        =>  VD_O( 31 downto 0 ),
        VLOAD       =>  VLOAD
```

```
);

end architecture VMECOREC1;
```

WISHBONE
COMPATIBLE

Specification for the:

WISHBONE System-on-Chip (SoC)
Interconnection Architecture
for Portable IP Cores

Revision: B.2, Released: October 10, 2001

Preliminary

Silicore

Electronic Design
Sensors • IP Cores

This Page is Intentionally Blank

Stewardship

Stewardship for this specification is maintained by Silicore Corporation. Questions, comments and suggestions about this document are welcome and should be directed to:

Wade D. Peterson, Silicore Corporation
6310 Butterworth Lane – Corcoran, MN USA 55340
TEL: (763) 478-3567; FAX: (763) 478-3568
E-MAIL: wadep@silicore.net URL: www.silicore.net

Silicore Corporation maintains this document as a service to its customers and to the IP core industry as a whole. The intent of this specification is to improve the quality of Silicore products, as well as to foster cooperation among the users and suppliers of IP cores.

Copyright & Trademark Release / Royalty Release / Patent Notice

Disclaimers

Document Format, Binding and Covers

This document is formatted for printing on double sided, 8½" x 11" white paper stock. It is designed to be bound within a standard cover. The preferred binding method is a black coil binding with outside diameter of 9/16" (14.5 mm). The preferred cover stock is Paper Direct part number KVR09D (forest green) and is available on-line at: www.paperdirect.com. Binding can be performed at most full-service copy centers such as Kinkos (www.kinkos.com).

Acknowledgements

Revision History

The various revisions of the WISHBONE specification, along with their changes and revision history, can be found at www.silicore.net/wishbone.htm.

Table of Contents

Chapter 1 - Introduction

The WISHBONE[1] System-on-Chip (SoC) Interconnection Architecture for Portable IP Cores is a flexible design methodology for use with semiconductor IP cores. Its purpose is to foster design reuse by alleviating System-on-Chip integration problems. This is accomplished by creating a common interface between IP cores. This improves the portability and reliability of the system, and results in faster time-to-market for the end user.

Previously, IP cores used non-standard interconnection schemes that made them difficult to integrate. This required the creation of custom glue logic to connect each of the cores together. By adopting a standard interconnection scheme, the cores can be integrated more quickly and easily by the end user.

This specification can be used for soft core, firm core or hard core IP. Since firm and hard cores are generally conceived as soft cores, the specification is written from that standpoint.

This specification does not require the use of specific development tools or target hardware. Furthermore, it is fully compliant with virtually all logic synthesis tools. However, the examples presented in the specification do use the VHDL hardware description language. These are presented only as a convenience to the reader, and should be readily understood by users of other hardware description languages (such as Verilog®). Schematic based tools can also be used.

The WISHBONE interconnect is intended as a general purpose interface. As such, it defines the standard data exchange between IP core modules. It does not attempt to regulate the application-specific functions of the IP core.

The WISHBONE architects were strongly influenced by three factors. First, there was a need for a good, reliable System-on-Chip integration solution. Second, there was a need for a common interface specification to facilitate structured design methodologies on large project teams. Third, they were impressed by the traditional system integration solutions afforded by micro-computer buses such as PCI bus and VMEbus.

In fact, the WISHBONE architecture is analogous to a microcomputer bus in that that they both: (a) offer a flexible integration solution that can be easily tailored to a specific application; (b) offer a variety of bus cycles and data path widths to solve various system problems; and (c) allow products to be designed by a variety of suppliers (thereby driving down price while improving performance and quality).

---

[1] Webster's dictionary defines a WISHBONE as "the forked clavicle in front of the breastbone of most birds." The term 'WISHBONE interconnect' was coined by Wade Peterson of Silicore Corporation. During the initial definition of the scheme he was attempting to find a name that was descriptive of a bi-directional data bus that used either multiplexors or three-state logic. This was solved by forming an interface with separate input and output paths. When these paths are connected to three-state logic it forms a 'Y' shaped configuration that resembles a wishbone. The actual name was conceived during a Thanksgiving Day dinner that included roast turkey. Thanksgiving Day is a national holiday in the United States, and is observed on the third Thursday in November. It is generally celebrated with a traditional turkey dinner.

However, traditional microcomputer buses are fundamentally handicapped for use as a System-on-Chip interconnection. That's because they are designed to drive long signal traces and connector systems which are highly inductive and capacitive. In this regard, System-on-Chip is much simpler and faster. Furthermore, the System-on-Chip solutions have a rich set of interconnection resources. These do not exist in microcomputer buses because they are limited by IC packaging and mechanical connectors.

The WISHBONE architects have attempted to create a specification that is robust enough to insure complete compatibility between IP cores. However, it has not been over specified so as to unduly constrain the creativity of the core developer or the end user. It is believed that these two goals have been accomplished with the publication of this document.

1.1 WISHBONE Features

The WISHBONE interconnection makes System-on-Chip and design reuse easy by creating a standard data exchange protocol. Features of this technology include:

- Simple, compact, logical IP core hardware interfaces that require very few logic gates.

- Supports structured design methodologies used by large project teams.

- Full set of popular data transfer bus protocols including:

    - READ/WRITE cycle
    - BLOCK transfer cycle
    - RMW cycle

- Data bus widths[2] and operand sizes up to 64-bits.

- Supports both BIG ENDIAN and LITTLE ENDIAN data ordering.

- Variable core interconnection methods support point-to-point, shared bus, crossbar switch, and switched fabric interconnections.

- Handshaking protocol allows each IP core to throttle its data transfer speed.

- Supports single clock data transfers.

- Supports normal cycle termination, retry termination and termination due to error.

- Address widths[3] up to 64-bits.

---

[2] Specifications are given for data port and operand sizes up to 64-bits. However, the basic architecture can theoretically support any data width (e.g. 128-bit, 256-bit etc.). Also, zero bit data bus accesses are permissible (generally used in FIFO interfaces).

o Partial address decoding scheme for SLAVEs. This facilitates high speed address decoding, uses less redundant logic and supports variable address sizing and interconnection means.

o User-defined tag support. This is useful for identifying transfers such as:

  - Data transfers
  - Interrupt vectors
  - Cache control operations

o MASTER / SLAVE architecture for very flexible system designs.

o Multiprocessing (multi-MASTER) capabilities. This allows for a wide variety of System-on-Chip configurations.

o Arbitration methodology is defined by the end user (priority arbiter, round-robin arbiter, etc.).

o Supports various IP core interconnection means, including:

  - Unidirectional bus
  - Bi-directional bus
  - Multiplexor based interconnections
  - Three-state based interconnections
  - Off chip I/O

o Synchronous design assures portability, simplicity and ease of use.

o Very simple, variable timing specification.

o Documentation requirements allow the end user to quickly evaluate interface needs.

o Independent of hardware technology (FPGA, ASIC, etc.).

o Independent of delivery method (soft, firm or hard core).

o Independent of synthesis tool, router and layout tool technology.

o Independent of FPGA and ASIC test methodologies.

o Seamless design progression between FPGA prototypes and ASIC production chips.

---

[3] Specifications are given for address widths between zero and 64-bits. However, the basic architecture can theoretically support any address width.

1.2 WISHBONE Objectives

The main objective of the specification is to create a flexible interconnection means for use with semiconductor IP cores. This allows various IP core modules to be connected together to form a System-on-Chip.

A further objective of the specification is to enforce good compatibility between IP core modules. This enhances design reuse.

A further objective of the specification is to create a robust standard, but one that does not unduly constrain the creativity of the core developer or the end user.

A further objective of the specification is to make it easy to understand by both the core developer and the end user.

A further objective of the specification is to facilitate structured design methodologies on large project teams. With structured design, individual team members can build and test small parts of the design. Each member of the design team can interface to the common, well-defined WISHBONE specification. When all of the sub-assemblies have been completed, the full system can be integrated.

A further objective of the specification is create a portable interface that is independent of the underlying semiconductor technology. For example, the interconnect must be capable of working with both FPGA and ASIC hardware target devices.

A further objective of the specification is to make the interface independent of logic signaling levels.

A further objective of the specification is to create a flexible interconnection scheme that is independent of the IP core delivery method. For example, it may be used with 'soft core', 'firm core' or 'hard core' delivery methods.

A further objective of the specification is to be independent of the underlying hardware description. For example, soft cores may be written and synthesized in VHDL, Verilog® or some other hardware description language. Schematic entry may also be used.

A further objective of the specification is to require a minimum standard for documentation. This allows IP core users to quickly evaluate and integrate new cores.

A further objective of the specification is to eliminate extensive interface documentation on the part of the IP core developer. In most cases, this specification along with the WISHBONE DATASHEET is sufficient to completely document an IP core data interface.

A further objective of the specification is to identify critical System-on-Chip interconnection technologies, and to place them into the public domain at the earliest possible date. This makes

it more difficult for individuals and organizations to create proprietary SoC technologies through the use of patent, trademark, copyright and trade secret protection mechanisms. This objective applies only to the interconnection of IP cores, but not to the IP cores themselves.

A further objective is to create an architecture that has a smooth transition path to support new technologies. This increases the longevity of the specification as it can adapt to new, and as yet un-thought-of, requirements.

A further objective is to create an architecture that allows various interconnection means between IP core modules. This insures that the end user can tailor the System-on-Chip to his/her own needs. For example, the entire interconnection system (which is analogous to a backplane on a standard microcomputer bus like VMEbus or cPCI) can be created by the system integrator. This allows the interconnection to be tailored to the final target device.

A further objective is to create an architecture that requires a minimum of glue logic. In some cases the System-on-Chip needs no glue logic whatsoever. However, in other cases the end user may choose to use a more sophisticated interconnection method (for example with FIFO memories or crossbar switches) that requires additional glue logic.

A further objective is to create an architecture with variable address and data path widths to meet a wide variety of system requirements.

A further objective is to create an architecture that fully supports the automatic generation of interconnection systems. This allows the WISHBONE interconnection to be generated with parametric core generators.

A further objective is to create an architecture that supports both BIG ENDIAN and LITTLE ENDIAN data transfer organizations.

A further objective is to create an architecture that supports one data transfer per clock cycle.

A further objective is to create a flexible architecture that allows data to be tagged. TAGs are user defined signals that allow each IP core to communicate with the rest of the system. They are especially useful when novel or unusual control signals (such as parity, cache control or interrupt acknowledge) are needed on an interface.

A further objective is to create an architecture with a MASTER/SLAVE topology. Furthermore, the system must be capable of supporting multiple MASTERs and multiple SLAVEs with an efficient arbitration mechanism.

A further objective is to create an architecture that supports point-to-point interconnections between IP cores.

A further objective is to create an architecture that supports shared bus interconnections between IP cores.

A further objective is to create an architecture that supports crossbar switches between IP cores.

A further objective is to create an architecture that supports switched fabrics.

A further objective is to create a synchronous protocol to insure ease of use, good reliability and easy testing. Furthermore, all transactions can be coordinated by a single clock.

A further objective is to create a synchronous protocol that works over a wide range of interface clock speeds. The effects of this are: (a) that the WISHBONE interface can work synchronously with all attached IP cores, (b) that the interface can be used on a large range of target devices, (c) that the timing specification is much simpler and (d) that the resulting semiconductor device is much more testable.

A further objective is to create a variable timing mechanism whereby the system clock frequency can be adjusted so as to control the power consumption of the integrated circuit.

A further objective is to create a synchronous protocol that provides a simple timing specification. This makes the interface very easy to integrate.

A further objective is to create a synchronous protocol where each MASTER and SLAVE can throttle the data transfer rate with a handshaking mechanism.

A further objective is to create a synchronous protocol that is optimized for System-on-Chip, but that is also suitable for off-chip I/O routing. Generally, the off-chip WISHBONE interconnect will operate at slower speeds.

1.3 Specification Terminology

To avoid confusion, and to clarify the requirements for compliance, this specification makes use of five keywords to define the operation of the WISHBONE interconnect. The keywords are:

- **RULE**
- **RECOMMENDATION**
- **SUGGESTION**
- **PERMISSION**
- **OBSERVATION**

Any text not labeled with one of these keywords describes the operation in a narrative style. The keywords are defined as follows:

RULE
Rules form the basic framework of the specification. They are sometimes expressed in text form and sometimes in the form of figures, tables or drawings. All rules MUST be followed to ensure compatibility between interfaces. Rules are characterized by an imperative style. The upper-

case words MUST and MUST NOT are reserved exclusively for stating rules in this document, and are not used for any other purpose.

RECOMMENDATION
Whenever a recommendation appears, designers would be wise to take the advice given. Doing otherwise might result in some awkward problems or poor performance. While this specification has been designed to support high performance systems, it is possible to create an interconnection that complies with all the rules, but has very poor performance. In many cases a designer needs a certain level of experience with the system architecture in order to design interfaces that deliver top performance. Recommendations found in this document are based on this kind of experience and are provided as guidance for the user.

SUGGESTION
A suggestion contains advice which is helpful but not vital. The reader is encouraged to consider the advice before discarding it. Some design decisions are difficult until experience has been gained. Suggestions help a designer who has not yet gained this experience. Some suggestions have to do with designing compatible interconnections, or with making system integration easier.

PERMISSION
In some cases a rule does not specifically prohibit a certain design approach, but the reader might be left wondering whether that approach might violate the spirit of the rule, or whether it might lead to some subtle problem. Permissions reassure the reader that a certain approach is acceptable and will not cause problems. The upper-case word MAY is reserved exclusively for stating a permission and is not used for any other purpose.

OBSERVATION
Observations do not offer any specific advice. They usually clarify what has just been discussed. They spell out the implications of certain rules and bring attention to things that might otherwise be overlooked. They also give the rationale behind certain rules, so that the reader understands why the rule must be followed.

1.4 Use of Timing Diagrams

Figure 1-1 shows some of the key features of the timing diagrams in this specification. Unless otherwise noted, the MASTER signal names are referenced in the timing diagrams. In some cases the MASTER and SLAVE signal names are different. For example, in the single MASTER / single SLAVE configuration, the [ADR_O] and [ADR_I] signals are connected together. Furthermore, the actual waveforms at the SLAVE may vary from those at the MASTER. That's because the MASTER and SLAVE interfaces can be connected together in different ways. Unless otherwise noted, the timing diagrams refer to the connection diagram shown in Figure 1-2.
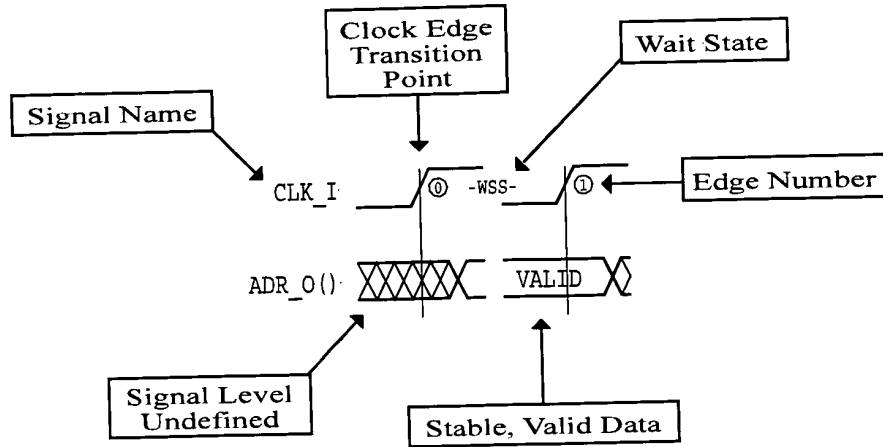
Figure 1-1. Use of timing diagrams.
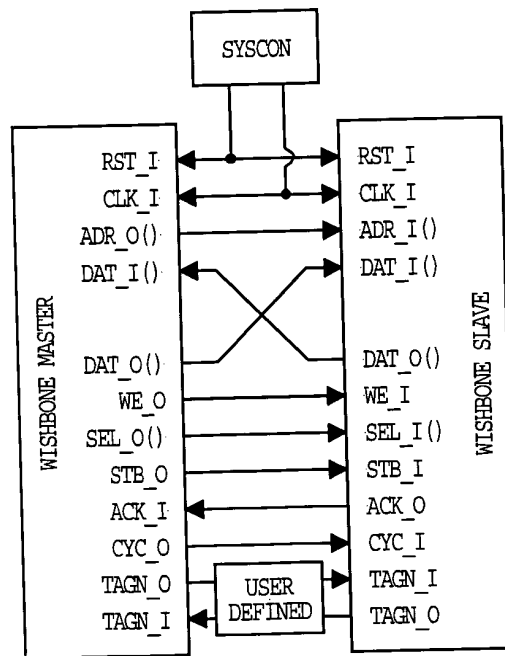
Figure 1-2. Standard connection for timing diagrams.

Individual signals may or may not be present on an specific interface. That's because many of the signals are optional.

Two symbols are also presented in relation to the [CLK_I] signal. These include the positive going clock edge transition point and the clock edge number. In most diagrams a vertical guide-line is shown at the positive-going edge of each [CLK_I] transition. This represents the theoretical transition point at which flip-flops register their input value, and transfer it to their output. The exact level of this transition point varies depending upon the technology used in the target device. The clock edge number is included as a convenience so that specific points in the timing diagram may be referenced in the text. The clock edge number in one timing diagram is not related to the clock edge number in another diagram.

Gaps in the timing waveforms may be shown. These indicate either: (a) a wait state or (b) a portion of the waveform that is not of interest (in the context of the diagram). When the gap indicates a wait state, the symbols '-WSM-' or '-WSS-' are placed in the gap along the [CLK_I] waveform. These correspond to wait states inserted by the MASTER or SLAVE interfaces respectively. They also indicate that the signals (with the exception of clock transitions and hatched regions) will remain in a steady state during that time.

Undefined signal levels are indicated by a hatched region. This region indicates that the signal level is undefined, and may take any state. It also indicates that the current state is undefined, and should not be relied upon. When signal arrays are used, stable and predictable signal levels are indicated with the word 'VALID'.

1.5 Signal Naming Conventions

All signal names used in this specification have the '_I' or '_O' characters attached to them. These indicate if the signals are an input (to the core) or an output (from the core). For example, [ACK_I] is an input and [ACK_O] is an output. This convention is used to clearly identify the direction of each signal.

In some cases, the input and output characters 'I' and 'O' may be omitted and replaced by an 'X'. For example: [TAG3_X]. This is not an actual signal name, but rather a shorthand form to indicate both the [TAG3_I] and [TAG3_O] signal.

Signal arrays are identified by a name followed by the array boundaries in parenthesis. For example, [DAT_I(63..0)] is a signal array with upper array boundary number sixty-three, and lower array boundary number zero. Furthermore, the array boundaries indicate the full range of the permissible array size. The array size on any particular core may vary. In many cases the array boundaries are omitted if they are irrelevant to the context of the description.

When used as part of a sentence, signal names are enclosed in brackets '[ ]'. This helps to discriminate signal names from the words in the sentence.

1.6 WISHBONE Logo

The WISHBONE logo can be affixed to SoC documents that are compatible with this standard. Figure 1-3 shows the logo.

Figure 1-3. WISHBONE logo.

PERMISSION 1.00
Documents describing a WISHBONE compatible SoC component that are 100% compliant with this standard, MAY use the WISHBONE logo.

1.7 Glossary of Terms

0x (numerical prefix)
The '0x' prefix indicates a hexadecimal number. It is the same nomenclature as commonly used in the 'C' programming language.

Active High Logic State
A logic state that is 'true' when the logic level is a binary '1'. The high state is at a higher voltage than the low state.

Active Low Logic State
A logic state that is 'true' when the logic level is a binary '0'. The low state is at a lower voltage than the high state.

ASIC
Acronym for: Application Specific Integrated Circuit. General term which describes a generic array of logic gates or analog building blocks which are programmed by a metalization layer at a silicon foundry. High level circuit descriptions are impressed upon the logic gates or analog building blocks in the form of metal interconnects.

Asserted
(1) A verb indicating that a logic state has switched from the inactive to the active state. When active high logic is used it means that a signal has switched from a logic low level to a logic high level. (2) *Assert:* to cause a signal line to make a transition from its logically false (inactive) state to its logically true (active) state. Opposite of *negated.*

Bus

(1) A common group of signals. (2) A signal line or a set of lines used by a data transfer system to connect a number of devices.

Bus Interface

An electronic circuit that drives or receives data or power from a bus.

Bus Cycle

The process whereby digital signals effect the transfer of data across a bus by means of an interlocked sequence of control signals. Also see: *Phase (bus cycle)*.

Crossbar Interconnect (Switch)

Crossbar switches are mechanisms that allow modules to connect and communicate. Each connection channel can be operated in parallel to other connection channels. This increases the data transfer rate of the entire system by employing parallelism. Stated another way, two 100 Mbyte/second channels can operate in parallel, thereby providing a 200 Mbyte/second transfer rate. This makes the crossbar switches inherently faster than traditional bus schemes. Crossbar routing mechanisms generally support dynamic configuration. This creates a configurable and reliable network system. Most crossbar architectures are also scalable, meaning that families of crossbars can be added as the needs arise. A crossbar interconnection is shown in Figure 1-4.
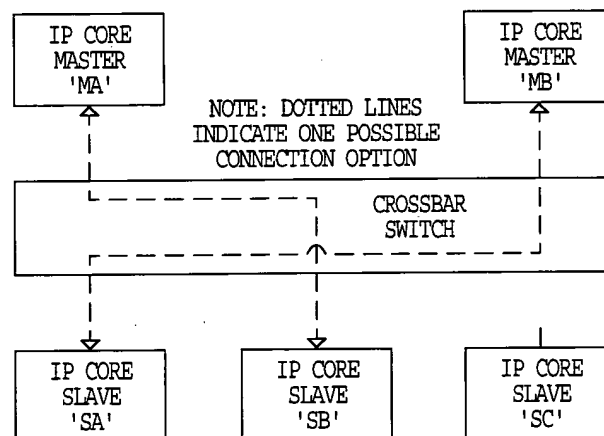
Figure 1-4. Crossbar (switch) interconnection.

Data Flow Interconnection

An interconnection where data flows through a prearranged set of IP cores in a sequential order. Data flow architectures often have the advantage of parallelism, whereby two or more functions are executed at the same time. Figure 1-5 shows a data flow interconnection between IP cores.

Figure 1-5. Data flow interconnection.

Data Organization

The ordering of data during a transfer. Generally, 8-bit (byte) data can be stored with the most significant byte of a mult-byte transfer at the higher or the lower address. These two methods are generally called BIG ENDIAN and LITTLE ENDIAN, respectively. In general, BIG ENDIAN refers to byte lane ordering where the most significant byte is stored at the lower address. LITTLE ENDIAN refers to byte lane ordering where the most significant byte is stored at the higher address. The terms BIG ENDIAN and LITTLE ENDIAN for data organization was coined by Danny Cohen of the Information Sciences Institute, and was derived from the book Gulliver's Travels by Jonathan Swift.

DMA Unit

Acronym for Direct Memory Access Unit. (1) A device that transfers data from one location in memory to another location in memory. (2) A device for transferring data between a device and memory without interrupting program flow. (3) A device that does not use low-level instructions. Intended for transferring data between memory or I/O locations.

ENDIAN

See the definition under 'Data Organization'.

FIFO

Acronym for: First In First Out. A type of memory used to transfer data between ports on two devices. In FIFO memories, data is removed in the same order that they were added. The FIFO memory is very useful for interconnecting cores of differing speeds.

Firm Core

An IP Core that is delivered in a way that allows conversion into an integrated circuit design, but does not allow the design to be easily reverse engineered. It is analogous to a binary or object file in the field of computer software design.

Fixed Interconnection
An interconnection system that is fixed, and *cannot* be changed without causing incompatibilities between bus modules (or SoC/IP cores). Also called a *static interconnection*. Examples of fixed interconnection buses include PCI, cPCI and VMEbus. Also see: *variable interconnection*.

Fixed Timing Specification
A timing specification that is based upon a fixed set of rules. Generally used in traditional microcomputer buses like PCI and VMEbus. Each bus module must conform to the ridged set of timing specifications. Also see: *variable timing specification*.

Foundry
See silicon foundry.

FPGA
Acronym for: Field Programmable Gate Array. Describes a generic array of logical gates and interconnect paths which are programmed by the end user. High level logic descriptions are impressed upon the gates and interconnect paths, often in the form of IP Cores.

Full Address Decoding
A method of address decoding where each SLAVE decodes all of the available address space. For example, if a 32-bit address bus is used, then each SLAVE decodes all thirty-two address bits. This technique is used on standard microcomputer buses like PCI and VMEbus. Also see: *partial address decoding*.

Gated Clock
A type of SYSCON interface where clock signal [CLK_O] can be stopped and restarted. The signal is always stopped in its low state. This technique is often used to reduce the power consumption of an integrated circuit. Under WISHBONE, the gated clock generator is optional. Also see: *variable clock generator*.

Glue Logic
(1) Logic gates and interconnections required to connect IP cores together. The requirements for glue logic vary greatly depending upon the interface requirements of the IP cores. (2) A family of logic circuits consisting of various gates and simple logic elements, each of which serve as an interface between various parts of a computer system.

Granularity
The smallest unit of data transfer that a port is capable of transferring. For example, a 32-bit port can be broken up into four 8-bit BYTE segments. In this case, the granularity of the interface is 8-bits. Also see: *port size* and *operand size*.

Hard Core
An IP Core that is delivered in the form of a mask set (i.e. a graphical description of the features and connections in an integrated circuit).

Hardware Description Language (HDL)
(1) Acronym for: Hardware Description Language. Examples include VHDL and Verilog®. (2) A general-purpose language used for the design of digital electronic systems.

Interface
A combination of signals and data-ports on a module which are capable of either generating, receiving, controlling or interconnecting IP cores. WISHBONE defines these as MASTER, SLAVE, INTERCON and SYSCON interfaces respectively. Also see: *MASTER, SLAVE, INTERCON* and *SYSCON.*

INTERCON
A WISHBONE interface that interconnects MASTER, SLAVE and SYSCON interfaces.

IP Core
Acronym for: Intellectual Property Core. Also see: *soft core, firm core* and *hard core.*

Mask Set
A graphical description of the features and connections in an integrated circuit.

MASTER
A WISHBONE interface that is capable of generating bus cycles. All systems based on the WISHBONE interconnect must have at least one MASTER interface. Also see: *SLAVE, SYSCON* and *INTERCON.*

Memory Mapped Addressing
An architecture that allows data to be stored and recalled in memory at individual, binary addresses.

Minimization (Logic Minimization)
A process by which HDL synthesis, router or other software development tools remove unused logic. This is important in WISHBONE because there are optional signals defined on many of the interfaces. If a signal is unused, then the logic minimization tools will remove these signals and their associated logic, thereby making a faster and more efficient design.

Module
In the context of this specification, it's another name for an IP core.

Multiplexor Interconnection
An interconnection that uses multiplexors to route address, data and control signals. Often used for System-on-Chip (SoC) applications. Also see: *three-state bus interconnection.*

Negated
A verb indicating that a logic state has switched from the active to the inactive state. When active high logic is used it means that a signal has switched from a logic high level to a logic low level. Also see: *asserted.*

Off-Chip Interconnection

An off-chip interconnection is used when a WISHBONE interface extends off-chip. See Figure 1-6.
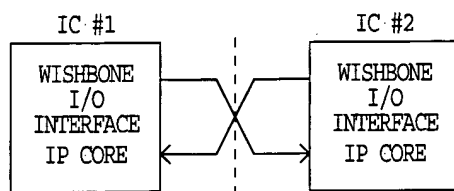
Figure 1-6. Off-chip interconnection.

Operand Size

The operand size is the largest single unit of data that is moved through an interface. For example, a 32-bit DWORD operand can be moved through an 8-bit port with four data transfers. Also see: *granularity* and *port size*.

Parametric Core Generator

A software tool used for the generation of IP cores based on input parameters. One example of a parametric core generator is a DSP filter generator. These are programs that create lowpass, bandpass and highpass DSP filters. The parameters for the filter are provided by the user, which causes the program to produce the digital filter as a VHDL or Verilog® hardware description. Parametric core generators can also be used create WISHBONE interconnections.

Partial Address Decoding

A method of address decoding where each SLAVE decodes only the range of addresses that it requires. For example, if the module needs only four addresses, then it decodes only the two least significant address bits. The remaining address bits are decoded by the interconnection system. This technique is used on SoC buses, and has the advantages of: less redundant logic in the system, it supports variable address buses, it supports variable interconnection buses and is relatively fast. Also see: *full address decoding*.

PCI

Acronym for: Peripheral Component Interconnect. Generally used as an interconnection scheme between integrated circuits. It also exists as a board level interconnection known as Compact PCI (or cPCI). While this specification is very flexible, it isn't practical for SoC applications.

Phase (Bus Cycle)

A periodic portion of a bus cycle. For example, a WISHBONE BLOCK READ cycle could contain ten phases, with each phase transferring a single 32-bit word of data. Collectively, the ten phases form the BLOCK READ cycle.

Point-to-point Interconnection

(1) An interconnection system that supports a single WISHBONE MASTER and a single WISHBONE SLAVE interface. It is the simplest way to connect two cores. See Figure 1-7. (2) A connection with only two endpoints.
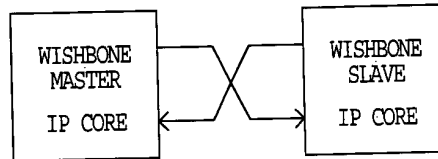
Figure 1-7. Point to point interconnection.

Port Size

The width of the WISHBONE data ports in bits. Also see: *granularity* and *operand size*.

Router

A software tool that physically routes interconnection paths between logic gates. Applies to both FPGA and ASIC devices.

RTL

(1) Register-transfer logic. A design methodology that moves data between registers. Data is latched in the registers at one or more stages along the path of signal propagation. The WISHBONE specification uses a synchronous RTL design methodology that requires that each register be clocked with a common clock. (2) Register-transfer level. A description of computer operations where data transfers from register to register, latch to latch and through logic gates. (3) A level of description of a digital design in which the clocked behavior of the design is expressly described in terms of data transfers between storage elements, which may be implied, and combinational logic, which may represent any computing or arithmetic-logic-unit logic. RTL modeling allows design hierarchy that represents a structural description of other RTL models.

Shared Bus Interconnection

The shared bus interconnection is a system where a MASTER initiates addressable bus cycles to a target SLAVE. Traditional buses such as VMEbus and PCI bus use this type of interconnection. As a consequence of this architecture, only one MASTER at a time can use the interconnection resource (i.e. bus). Figure 1-8 shows an example of a WISHBONE shared bus interconnection.
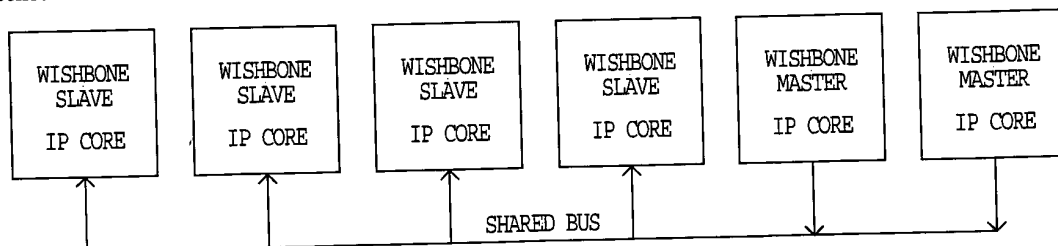
Figure 1-8. Shared bus interconnection.

Silicon Foundry
A factory that produces integrated circuits.

SLAVE
A WISHBONE interface that is capable of receiving bus cycles. All systems based on the WISHBONE interconnect must have at least one SLAVE. Also see: *MASTER, SYSCON* and *INTERCON.*

Soft Core
An IP Core that is delivered in the form of a hardware description language or schematic diagram.

SoC
Acronym for System-on-Chip. Also see: *System-on-Chip.*

Structured Design
(1) A popular method for managing complex projects. Often used with large project teams. When structured design practices are used, individual team members build and test small parts of the design with a common set of tools. Each sub-assembly is designed to a common standard. When all of the sub-assemblies have been completed, the full system can be integrated and tested. This approach makes it much easier to manage the design process. (2) Any disciplined approach to design that adheres to specified rules based on principles such as modularity and top-down design.

Switched Fabric Interconnection
A type of interconnection that uses large numbers of crossbar switches. These are organized into arrays that resemble the threads in a fabric. The resulting system is a network of redundant interconnections.

SYSCON
A WISHBONE module and interface. The SYSCON *module* is the only module in the design which may contain the SYSCON *interface*. The SYSCON interface is the only interface in the design which may drive the system clock [CLK_O] and reset [RST_O] signals.

System-on-Chip (SoC)
A method by which whole systems are created on a single integrated circuit chip. In many cases, this requires the use of IP cores which have been designed by multiple IP core providers. System-on-Chip is similar to traditional microcomputer bus systems whereby the individual components are designed, tested and built separately. The components are then integrated to form a finished system.

Target Device
The semiconductor type (or technology) onto which the IP core design is impressed. Typical examples include FPGA and ASIC target devices.

Three-State Bus Interconnection
A microcomputer bus interconnection that relies upon three-state bus drivers. Often used to reduce the number of interconnecting signal paths through connector and IC pins. Three state buffers can assume a logic low state ('0' or 'L'), a logic high state ('1' or 'H') or a high impedance state. Three-state buffers are sometimes called Tri-State® buffers. Tri-State® is a registered trademark of National Semiconductor Corporation. Also see: *multiplexor interconnection.*

Variable Clock Generator
A type of SYSCON interface where the frequency of [CLK_O] can be changed dynamically. The frequency can be changed by way of a programmable phase-lock-loop (PLL) circuit or other control mechanism. This technique is used to reduce the power consumption of the circuit. The variable clock generator capability is optional. Also see: *gated clock generator* and *variable timing specification.*

Variable Interconnection
A microcomputer bus interconnection that *can* be changed without causing incompatibilities between bus modules (or SoC/IP cores). Also called a dynamic interconnection. An example of a variable interconnection bus is the WISHBONE SoC architecture. Also see: *fixed interconnection.*

Variable Timing Specification
A timing specification that is not fixed. In WISHBONE, variable timing can be achieved in a number of ways. For example, the system integrator can select the SYSCON frequency rate of [CLK_O] by enforcing a timing specification during the circuit design. Variable timing can also be achieved during circuit operation with a variable clock generator. Also see: *gated clock generator* and *variable clock generator.*

Verilog®
A textual based hardware description language (HDL) intended for use in circuit design. The Verilog® language is both a synthesis and a simulation tool. Verilog® was originally a proprietary language first conceived in 1983 at Gateway Design Automation (Acton, MA), and was later refined by Cadence Corporation. It has since been greatly expanded and refined, and much of it has been placed into the public domain. Complete descriptions of the language can be found in the IEEE 1364 specification.

VHDL
Acronym for: VHSIC Hardware Description Language. [VHSIC: Very High Speed Integrated Circuit]. A textual based computer language intended for use in circuit design. The VHDL language is both a synthesis and a simulation tool. Early forms of the language emerged from US Dept. of Defense ARPA projects in the 1960's, and have since been greatly expanded and refined. Complete descriptions of the language can be found in the IEEE 1076, IEEE 1073.3, IEEE 1164 specifications.

VMEbus
Acronym for: Versa Module Eurocard bus. A popular microcomputer (board) bus. While this specification is very flexible, it isn't practical for SoC applications.

WISHBONE DATASHEET
A type of documentation required for WISHBONE compatible IP cores. This helps the end user understand the detailed operation of the core, and how to connect it to other cores. The WISHBONE DATASHEET can be included as part of an IP core technical reference manual, or as part of the IP core hardware description.

WISHBONE Signal
A signal that is defined as part of the WISHBONE specification. Non-WISHBONE signals can also be used on the IP core, but are not defined as part of this specification. For example, [ACK_O] is a WISHBONE signal, but [CLK100_I] is not.

WISHBONE Logo
A logo that, when affixed to a document, indicates that the associated SoC component is compatible with the WISHBONE standard.

Wrapper
A circuit element that converts a non-WISHBONE IP Core into a WISHBONE compatible IP Core. For example, consider a 16-byte synchronous memory primitive that is provided by an IC vendor. The memory primitive can be made into a WISHBONE compatible SLAVE by layering a circuit over the memory primitive, thereby creating a WISHBONE compatible SLAVE. A wrapper is analogous to a technique used to convert software written in 'C' to that written in 'C++'.

1.8 References

IEEE 100: The Authoritative Dictionary of IEEE Standards Terms, Seventh Edition. IEEE Press 2000.

Chapter 2 – Interface Specification

This chapter describes the signaling method between MASTER, SLAVE, SYSCON and IN-TERCON interfaces. This includes numerous options which may or may not be present on a particular interface. Furthermore, it describes a minimum level of required documentation that must be created for each IP core.

2.1 Required Documentation for IP Cores

WISHBONE compatible IP cores must include documentation that describes the interface. This helps the end user understand the operation of the core, and how to connect it to other cores. This documentation takes the form of a WISHBONE DATASHEET. It can be included as part of the IP core technical reference manual, it can be embedded in source code or it can take other forms as well.

2.1.1 General Requirements for the WISHBONE DATASHEET

RULE 2.00
Each WISHBONE compatible IP core MUST include a WISHBONE DATASHEET as part of the IP core documentation.

RULE 2.05
The WISHBONE DATASHEET MUST include the revision level of the WISHBONE specification to which it was designed.

RULE 2.10
The WISHBONE DATASHEET MUST indicate whether it is a MASTER, SLAVE, SYSCON or INTERCON interface. Furthermore, it MUST indicate the types of bus cycles it supports.

RULE 2.15
The WISHBONE DATASHEET for MASTER, SLAVE and INTERCON interfaces MUST include the following information:

(1) If a MASTER supports the optional [ERR_I] signal, then the WISHBONE DA-TASHEET MUST describe how it reacts in response to the signal. If a SLAVE supports the optional [ERR_O] signal, then the WISHBONE DATASHEET MUST describe the conditions under which the signal is generated.

(2) If a MASTER supports the optional [RTY_I] signal, then the WISHBONE DA-TASHEET MUST describe how it reacts in response to the signal. If a SLAVE sup-

ports the optional [RTY_O] signal, then the WISHBONE DATASHEET MUST describe the conditions under which the signal is generated.

(3) All interfaces that support the [TAGN_I] or [TAGN_O] signal(s) MUST describe their use in the WISHBONE DATASHEET.

(4) The WISHBONE DATASHEET MUST indicate the port size. The port size MUST be indicated as: 8-bit, 16-bit, 32-bit or 64-bit.

(5) The WISHBONE DATASHEET MUST indicate the port granularity. The granularity MUST be indicated as: 8-bit, 16-bit, 32-bit or 64-bit.

(6) The WISHBONE DATASHEET MUST indicate the maximum operand size. The maximum operand size MUST be indicated as: 8-bit, 16-bit, 32-bit or 64-bit. If the maximum operand size is unknown, then the maximum operand size shall be the same as the granularity.

(7) The WISHBONE DATASHEET MUST indicate the data transfer ordering. The ordering MUST be indicated as BIG ENDIAN or LITTLE ENDIAN. When the port size equals the granularity, then the interface shall be specified as BIG ENDIAN and/or LITTLE ENDIAN. [When the port size equals the granularity, then BIG ENDIAN and LITTLE ENDIAN transfers are identical].

(8) The WISHBONE DATASHEET MUST indicate the sequence of data transfer through the port. If the sequence of data transfer is not known, then the datasheet MUST indicate that it is undefined.

(9) The WISHBONE DATASHEET MUST indicate if there are any constraints on the [CLK_I] signal. These constraints include (but are not limited to) clock frequency, application specific timing constraints, the use of gated clocks or the use of variable clock generators.

2.1.2 Signal Naming

RULE 2.20
Signal names MUST adhere to the rules of the native tool in which the IP core is designed.

PERMISSION 2.00
Any signal name MAY be used to describe the WISHBONE signals.

OBSERVATION 2.00
Most hardware description languages (such as VHDL or Verilog®) have naming conventions. For example, the VHDL hardware description language defines the alphanumeric symbols which

may be used. Furthermore, it states that UPPERCASE and LOWERCASE characters may be used in a signal name.

RECOMENDATION 2.00
It is recommended that the interface use the signal names defined in this document.

OBSERVATION 2.05
Core integration is simplified if the signal names match those given in this specification. However, in some cases (such as IP cores with multiple WISHBONE interconnects) they cannot be used. The use of non-standard signal names will not result in any serious integration problems since all hardware description tools allow signals to be renamed.

RULE 2.25
The WISHBONE DATASHEET MUST include the signal names that are defined for a WISHBONE SoC interface. If a signal name is different than defined in this specification then it MUST be cross-referenced to the corresponding signal name which is used in this specification.

2.1.3 Logic Levels

RULE 2.30
All WISHBONE interface signals MUST use active high logic.

OBSERVATION 2.10
In general, the use of active low signals does not present a problem. However, RULE 2.30 is included because some tools (especially schematic entry tools) do not have a standard way of indicating an active low signal. For example, a reset signal could be named [#RST_I], [/RST_I] or [N_RST_I]. This was found to cause confusion among users and incompatibility between modules. This constraint should not create any undue difficulties, as the system integrator can invert any signals before use by the WISHBONE interface.

PERMISSION 2.05
Non-WISHBONE signals MAY be used with IP core interfaces.

OBSERVATION 2.15
Most IP cores will include non-WISHBONE signals. These are outside the scope of this specification, and no attempt is made to govern them. For example, a disk controller IP core could have a WISHBONE interface on one end and a disk interface on the other. In this case the specification does not dictate any technical requirements for the disk interface signals.

OBSERVATION 2.20
[TAGN_I] and [TAGN_O] are user defined signals that must adhere to the timing specifications given in this document.

2.2 WISHBONE Signal Description

This section describes the signals used in the WISHBONE interconnect. Some of these signals are optional, and may or may not be present on a specific interface.

2.2.1 SYSCON Signals

CLK_O
The system clock output [CLK_O] is generated by the SYSCON interface. It coordinates all activities for the internal logic within the WISHBONE interconnect. The INTERCON connects the [CLK_O] output to the [CLK_I] input on MASTER and SLAVE interfaces.

RST_O
The reset output [RST_O] is generated by the SYSCON interface. It forces all WISHBONE interfaces to restart. All internal self-starting state machines are forced into an initial state. The INTERCON connects the [RST_O] output to the [RST_I] input on MASTER and SLAVE interfaces.

2.2.2 Signals Common to MASTER and SLAVE Interfaces

CLK_I
The clock input [CLK_I] coordinates all activities for the internal logic within the WISHBONE interconnect. All WISHBONE output signals are registered at the rising edge of [CLK_I]. All WISHBONE input signals must be stable before the rising edge of [CLK_I].

RST_I
The reset input [RST_I] forces the WISHBONE interface to restart. Furthermore, all internal self-starting state machines will be forced into an initial state. This signal only resets the WISHBONE interface. It is not required to reset other parts of an IP core (although it may be used that way).

TAGN_I
The tag input(s) [TAGN_I] are user defined, and are used with either MASTER or SLAVE interfaces. 'N' in this signal name refers to a tag number because multiple tags may be used (e.g.

[TAG3_I]). Tag inputs are used whenever an IP core needs specific information from the interconnection. For example, a MASTER can be designed to monitor the state of a FIFO.

TAGN_O
The tag output(s) [TAGN_O] are user defined, and are used with either MASTER or SLAVE interfaces. For example, the tag output(s) can be used to indicate the type of data transfer in progress. Furthermore, 'N' in this signal name refers to a tag number because multiple tags may be used. For example, [TAG1_O] may indicate a valid data transfer cycle, [TAG2_O] may indicate an interrupt acknowledge cycle and so on. The exact meaning of each tag is defined by the IP core provider in the WISHBONE DATASHEET.

2.2.3 MASTER Signals

ACK_I
The acknowledge input [ACK_I], when asserted, indicates the termination of a normal bus cycle. Also see the [ERR_I] and [RTY_I] signal descriptions.

ADR_O(63..0)
The address output array [ADR_O(63..0)] is used to pass a binary address, with the most significant address bit at the higher numbered end of the signal array. The lower array boundary is specific to the data port size. The higher array boundary is core-specific. In some cases (such as FIFO interfaces) the array may not be present on the interface.

CYC_O
The cycle output [CYC_O], when asserted, indicates that a valid bus cycle is in progress. The signal is asserted for the duration of all bus cycles. For example, during a BLOCK transfer cycle there can be multiple data transfers. The [CYC_O] signal is asserted during the first data transfer, and remains asserted until the last data transfer. The [CYC_O] signal is useful for interfaces with multi-port interfaces (such as dual port memories). In these cases, the [CYC_O] signal requests use of a common bus from an arbiter. Once the arbiter grants the bus to the MASTER, it is held until [CYC_O] is negated.

DAT_I(63..0)
The data input array [DAT_I(63..0)] is used to pass binary data. The array boundaries are determined by the port size. Also see the [DAT_O(63..0)] and [SEL_O(7..0)] signal descriptions.

DAT_O(63..0)
The data output array [DAT_O(63..0)] is used to pass binary data. The array boundaries are determined by the port size. Also see the [DAT_I(63..0)] and [SEL_O(7..0)] signal descriptions.

ERR_I
The error input [ERR_I] indicates an abnormal cycle termination. The source of the error, and the response generated by the MASTER is defined by the IP core supplier. Also see the [ACK_I] and [RTY_I] signal descriptions.

RTY_I
The retry input [RTY_I] indicates that the interface is not ready to accept or send data, and that the cycle should be retried. When and how the cycle is retried is defined by the IP core supplier. Also see the [ERR_I] and [RTY_I] signal descriptions.

SEL_O(7..0)
The select output array [SEL_O(7..0)] indicates where valid data is expected on the [DAT_I(63..0)] signal array during READ cycles, and where it is placed on the [DAT_O(63..0)] signal array during WRITE cycles. Also see the [DAT_I(63..0)], [DAT_O(63..0)] and [STB_O] signal descriptions.

STB_O
The strobe output [STB_O] indicates a valid data transfer cycle. It is used to qualify various other signals on the interface such as [SEL_O(7..0)]. The SLAVE must assert either the [ACK_I], [ERR_I] or [RTY_I] signals in response to every assertion of the [STB_O] signal.

WE_O
The write enable output [WE_O] indicates whether the current local bus cycle is a READ or WRITE cycle. The signal is negated during READ cycles, and is asserted during WRITE cycles.

2.2.4 SLAVE Signals

ACK_O
The acknowledge output [ACK_O], when asserted, indicates the termination of a normal bus cycle. Also see the [ERR_O] and [RTY_O] signal descriptions.

ADR_I(63..0)
The address input array [ADR_I(63..0)] is used to pass a binary address, with the most significant address bit at the higher numbered end of the signal array. The lower array boundary is specific to the data port size. The higher array boundary is core-specific. In some cases (such as FIFO interfaces) the array may not be present on the interface.

CYC_I

The cycle input [CYC_I], when asserted, indicates that a valid bus cycle is in progress. The signal is asserted for the duration of all bus cycles. For example, during a BLOCK transfer cycle there can be multiple data transfers. The [CYC_I] signal is asserted during the first data transfer, and remains asserted until the last data transfer.

DAT_I(63..0)

The data input array [DAT_I(63..0)] is used to pass binary data. The array boundaries are determined by the port size. Also see the [DAT_O(63..0)] and [SEL_O(7..0)] signal descriptions.

DAT_O(63..0)

The data output array [DAT_O(63..0)] is used to pass binary data. The array boundaries are determined by the port size. Also see the [DAT_I(63..0)] and [SEL_O(7..0)] signal descriptions.

ERR_O

The error output [ERR_O] indicates an abnormal cycle termination. The source of the error, and the response generated by the MASTER is defined by the IP core supplier. Also see the [ACK_O] and [RTY_O] signal descriptions.

RTY_O

The retry output [RTY_O] indicates that the indicates that the interface is not ready to accept or send data, and that the cycle should be retried. When and how the cycle is retried is defined by the IP core supplier. Also see the [ERR_O] and [RTY_O] signal descriptions.

SEL_I(7..0)

The select input array [SEL_I(7..0)] indicates where valid data is placed on the [DAT_I(63..0)] signal array during WRITE cycles, and where it should be present on the [DAT_O(63..0)] signal array during READ cycles. Also see the [DAT_I(63..0)], [DAT_O(63..0)] and [STB_I] signal descriptions.

STB_I

The strobe input [STB_I], when asserted, indicates that the SLAVE is selected. A SLAVE shall respond to other WISHBONE signals only when this [STB_I] is asserted, except for the [RST_I] signal which should always be responded to. The SLAVE must assert either the [ACK_O], [ERR_O] or [RTY_O] signals in response to every assertion of the [STB_I] signal.

WE_I

The write enable input [WE_I] indicates whether the current local bus cycle is a READ or WRITE cycle. The signal is negated during READ cycles, and is asserted during WRITE cycles.

Chapter 3 – Bus Cycles

WISHBONE bus cycles are described in terms of their general operation, reset operation, handshaking protocol and the data organization during transfers. Additional requirements for bus cycles (especially those relating to the common clock) can be found in the timing specifications in Chapter 4.

3.1 General Operation

Each MASTER and SLAVE are interconnected with a set of signals that permit them to exchange data. For descriptive purposes these signals are cumulatively known as a *bus*, and are contained within a functional module called the INTERCON. Address, data and other information is impressed upon this bus in the form of *bus cycles*.

3.1.1 Reset Operation

All hardware interfaces must be initialized to a pre-defined state. This is accomplished with the reset signal [RST_O], which can be asserted at any time. It is also used for test simulation purposes by initializing all self-starting state machines and counters which may be used in the design. The reset signal [RST_O] is driven by the SYSCON interface. It is connected to the [RST_I] signal on all MASTER and SLAVE interface. Figure 3-1 shows the reset cycle.


```
NOTES:
(1) Reset cycles can be extended for any length of time.
(2) Self-starting state machines & counters reset themselves at
    the rising [CLK_I] edge following the assertion of [RST_I].
    On MASTERs, [STB_O] and [CYC_O] are negated at the same time.
(3) On MASTERs, [STB_O] and [CYC_O] may be asserted at the
    rising [CLK_I] edge following the negation of [RST_I].
```
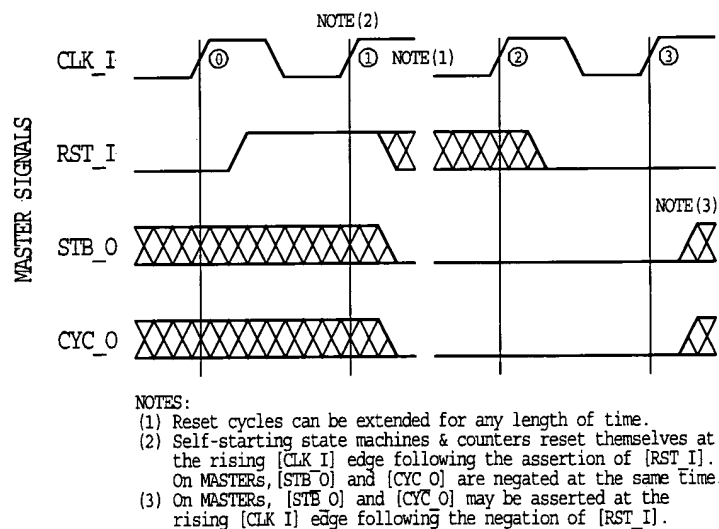
Figure 3-1. Reset cycle.

RULE 3.00

All WISHBONE interfaces MUST initialize themselves at the rising [CLK_O] edge following the assertion of [RST_O]. They MUST stay in the initialized state until the rising [CLK_O] edge that follows the negation of [RST_O].

RULE 3.05

[RST_I] MUST be asserted for at least one complete clock cycle on all WISHBONE interfaces.

PERMISSION 3.00

[RST_O] MAY be asserted for more than one clock cycle, and MAY be asserted indefinitely.

RULE 3.10

All WISHBONE interfaces MUST be capable of reacting to [RST_I] at any time.

RULE 3.15

All self-starting state machines and counters in WISHBONE interfaces MUST initialize themselves at the rising [CLK_I] edge following the assertion of [RST_I]. They MUST stay in the initialized state until the rising [CLK_I] edge that follows the negation of [RST_I].

OBSERVATION 3.00

In general, self-starting state machines do not need to be initialized. However, this may cause problems because some simulators may not be sophisticated enough to find an initial starting point for the state machine. Furthermore, self-starting state machines can go through an indeterminate number of initialization cycles before finding their starting state, thereby making it difficult to predict their behavior at start-up time. The initialization rule prevents both problems by forcing all state machines to a pre-defined state in response to the assertion of [RST_I].

RULE 3.20

The following MASTER signals MUST be negated at the rising [CLK_I] edge following the assertion of [RST_I], and MUST stay in the negated state until the rising [CLK_I] edge that follows the negation of [RST_I]: [STB_O], [CYC_O]. The state of all other MASTER signals are undefined in response to a reset cycle.

OBSERVATION 3.05

On MASTER interfaces [STB_O] and [CYC_O] may be asserted beginning at the rising [CLK_I] edge following the negation of [RST_I].

OBSERVATION 3.10
SLAVE interfaces automatically negate [ACK_O], [ERR_O] and [RTY_O] when their [STB_I] is negated.

RECOMENDATION 3.00
Design SYSCON circuits so that they assert [RST_O] during a power-up condition. [RST_O] should remain asserted until all voltage levels and clock frequencies in the system are stabilized. When negating [RST_O], do so in a synchronous manner that conforms to this specification.

OBSERVATION 3.15
If a gated clock generator is used, and if the clock is stopped, then the WISHBONE interface is not capable of responding to its [RST_I] signal.

SUGGESTION 3.00
Some circuits require an *asynchronous* reset capability. If an IP core or other SoC component requires an asynchronous reset, then define it as a non-WISHBONE signal. This prevents confusion with the WISHBONE reset [RST_O] signal, which uses a purely synchronous protocol, and need be applied only to the WISHBONE interface.

OBSERVATION 3.20
All WISHBONE *interfaces* must respond to the reset signal. However, the IP Core connected to a WISHBONE interface does not necessarily need to respond to the reset signal.

3.1.2 Handshaking Protocol

All bus cycles use a handshaking protocol between the MASTER and SLAVE interfaces. As shown in Figure 3-2, the MASTER asserts [STB_O] when it is ready to transfer data. [STB_O] remains asserted until the SLAVE asserts one of the cycle terminating signals [ACK_I], [ERR_I] or [RTY_I]. At every rising edge of [CLK_I] the terminating signal is sampled. If it is asserted, then [STB_O] is negated. This gives both the MASTER and SLAVE interfaces the possibility to control the rate at which data is transferred.

If the SLAVE guarantees it can keep pace with all MASTER interfaces, and if the [ERR_I] and [RTY_I] signals are not used, then the [ACK_I] signal may be tied to the SLAVE's [STB_I] input. The interface will function normally under these circumstances.
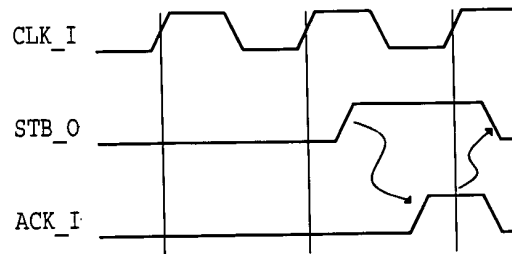
Figure 3-2. Local bus handshaking protocol.

Most of the examples in this specification describe the use of [ACK_I] to terminate a local bus cycle. However, the SLAVE can optionally terminate the cycle with an error [ERR_O], or request that the cycle be retried [RTY_O].

All interfaces include the [ACK_I] terminator signal. Asserting this signal during a bus cycle causes it to terminate normally.

Asserting the [ERR_I] signal during a bus cycle will terminate the cycle. It also serves to notify the MASTER that an error occurred during the cycle. This signal is generally used if an error was detected by SLAVE logic circuitry. For example, if the SLAVE is a parity-protected memory, then the [ERR_I] signal can be asserted if a parity fault is detected. This specification does not dictate what the MASTER will do in response to [ERR_I].

Asserting the optional [RTY_I] signal during a bus cycle will terminate the cycle. It also serves to notify the MASTER that the current cycle should be aborted, and retried at a later time. This signal is generally used for shared memory and bus bridges. In these cases SLAVE circuitry would assert [RTY_I] if the local resource is busy. This specification does not dictate when or how the MASTER will respond to [RTY_I].

RULE 3.25
As a minimum, the MASTER interface MUST include the following signals: [ACK_I], [CLK_I], [CYC_O], [RST_I] and [STB_O]. As a minimum, the SLAVE interface MUST include the following signals: [ACK_O], [CLK_I] and [RST_I]. All other signals are optional.

PERMISSION 3.05
MASTER and SLAVE interfaces MAY be designed to support the [ERR_I] and [ERR_O] signals. In these cases, the SLAVE asserts [ERR_O] to indicate that an error has occurred during the bus cycle. This specification does not dictate what the MASTER does in response to [ERR_I].

PERMISSION 3.10
MASTER and SLAVE interfaces MAY be designed to support the [RTY_I] and [RTY_O] signals. In these cases, the SLAVE asserts [RTY_O] to indicate that the interface is busy, and that the bus cycle should be retried at a later time. This specification does not dictate what the MASTER will do in response to [RTY_I].

RULE 3.30
If a SLAVE supports the [ERR_O] or [RTY_O] signals, then the SLAVE MUST NOT assert more than one of the following signals at any time: [ACK_O], [ERR_O] or [RTY_O].

RECOMMENDATION 3.00
Design WISHBONE MASTER interfaces so that there are no intermediate logic gates between a registered flip-flop and the signal outputs on [STB_O] and [CYC_O]. Delay timing for [STB_O] and [CYC_O] are very often the most critical paths in the system. This prevents sloppy design practices from slowing down the interconnect because of added delays on these two signals.

RULE 3.35
SLAVE interfaces MUST be designed so that the [ACK_O], [ERR_O] and [RTY_O] signals are asserted and negated in response to the assertion and negation of [STB_I]. Furthermore, this activity MUST occur asynchronous to the [CLK_I] signal (i.e. there is a combinatorial logic path between [STB_I] and [ACK_O], etc.).

OBSERVATION 3.25
The asynchronous logic requirement assures that the interface can accomplish one data transfer per clock cycle. Furthermore, it simplifies the design of arbiters in multi-MASTER applications.

RECOMMENDATION 3.05
Design interconnection logic to prevent deadlock conditions when MASTER accesses are made to unused address locations. One solution to this problem is to include a watchdog timer function that monitors the MASTER's [STB_O] signal, and asserts [ERR_I] or [RTY_I] if the cycle exceeds some pre-defined time limit.

PERMISSION 3.15
Under certain circumstances SLAVE interfaces MAY be designed to hold [ACK_O] in the asserted state. This situation occurs on point-to-point interfaces where there is a single SLAVE on the interface, and that SLAVE always operates without wait states.

RULE 3.40
MASTER interfaces MUST be designed to operate normally when SLAVE interface holds [ACK_I] in the asserted state.

3.1.3 Use of [STB_O]

RULE 3.45
MASTER interfaces MUST qualify the following signals with [STB_O]: [ADR_O], [DAT_O()], [SEL_O()], [WE_O], [SEL_O] and [TAGN_O].

RULE 3.50
MASTER interfaces MUST assert [CYC_O] for the duration of SINGLE READ / WRITE, BLOCK and RMW cycles. [CYC_O] MUST be asserted no later than the rising [CLK_I] edge that qualifies the assertion of [STB_O]. [CYC_O] MUST be negated no earlier than the rising [CLK_I] edge that qualifies the negation of [STB_O].

3.1.4 Use of [ACK_O], [ERR_O] and [RTY_O]

RULE 3.55
SLAVE interfaces MUST qualify the following signals with [ACK_O], [ERR_O] or [RTY_O]: [DAT_O()].

3.1.5 Use of [TAGN_I] and [TAGN_O] Signals

The TAG signals [TAGN_I] and [TAGN_O] are used by both the MASTER and SLAVE interfaces. They are used for three purposes: (a) to tag data with information such as parity or time stamps, (b) to identify specialty bus cycles (like interrupts or cache control operations) and (c) to communicate with the bus interconnection. These signals are user defined.

For example, the designer of a MASTER may wish to add parity check bits to its bus cycle. In this case a [TAGN_O] signal is defined by the IP core designer, and logic would be created to generate the bit. Furthermore, the signal would be described in the WISHBONE DATASHEET.

In another example, the designer of a SLAVE interface may wish to notify the bus interconnection logic with the size of it's data interface. In this case a [TAGN_O] signal is defined by the IP core designer, and logic would be created to reflect the bus size. The signal would also be described in the WISHBONE DATASHEET.

RULE 3.60
The [TAGN_I] and [TAGN_O] signals MUST adhere to the timing specifications given in this document.

3.2 SINGLE READ / WRITE Cycles

The SINGLE READ / WRITE cycles perform one data transfer at a time. These are the basic cycles used to perform data transfers on the WISHBONE interconnect.

RULE 3.65
All MASTER and SLAVE interfaces that support SINGLE READ or SINGLE WRITE cycles MUST conform to the timing requirements given in sections 3.2.1 and 3.2.2.

PERMISSION 3.20
MASTER and SLAVE interfaces MAY be designed so that they do not support the SINGLE READ or SINGLE WRITE cycles.

3.2.1 SINGLE READ Cycle

Figure 3-3 shows a SINGLE READ cycle. The bus protocol works as follows:

CLOCK EDGE 0: MASTER presents [ADR_O()] and [TAGN_O].
MASTER negates [WE_O] to indicate a READ cycle.
MASTER presents bank select [SEL_O()] to indicate where it expects data.
MASTER asserts [CYC_O] to indicate the start of the cycle.
MASTER asserts [STB_O] to qualify [ADR_O()], [SEL_O()] and [WE_O].

SETUP, EDGE 1: SLAVE decodes inputs, and responds by asserting [ACK_I].
SLAVE presents valid data on [DAT_I()].
SLAVE asserts [ACK_I] in response to [STB_O] to indicate valid data.
SLAVE presents [TAGN_O].
MASTER monitors [TAGN_I].
MASTER monitors [ACK_I], and prepares to latch data on [DAT_I()].

Note: SLAVE may insert wait states (-WSS-) before asserting [ACK_I], thereby allowing it to throttle the cycle speed. Any number of wait states may be added.

CLOCK EDGE 1: MASTER latches data on [DAT_I()].
MASTER latches [TAGN_I].
MASTER negates [STB_O] and [CYC_O] to indicate the end of the cycle.
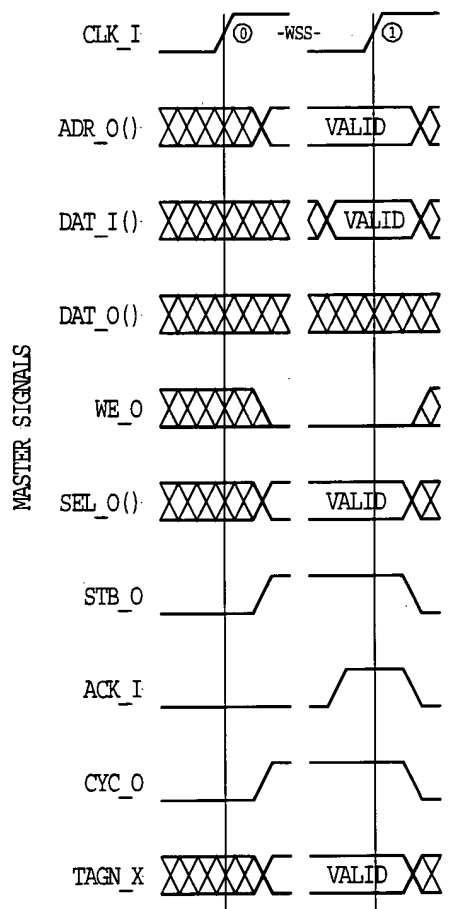
Figure 3-3. SINGLE READ cycle.

3.2.2 SINGLE WRITE Cycle

Figure 3-4 shows a SINGLE WRITE cycle. The bus protocol works as follows:

CLOCK EDGE 0: MASTER presents [ADR_O()] and [TAGN_O].
MASTER asserts [WE_O] to indicate a WRITE cycle.
MASTER presents bank select [SEL_O()] to indicate where it sends data.
MASTER asserts [CYC_O] to indicate the start of the cycle.
MASTER asserts [STB_O] to qualify [ADR_O()], [SEL_O()] and [WE_O].

SETUP, EDGE 1: SLAVE decides inputs, and responds by asserting [ACK_I].
SLAVE presents prepares to latch data on [DAT_O()].
SLAVE asserts [ACK_I] in response to [STB_O] to indicate latched data.
SLAVE presents [TAGN_O].
MASTER monitors [TAGN_I].
MASTER monitors [ACK_I], and prepares to terminate the cycle.

Note: SLAVE may insert wait states (-WSS-) before asserting [ACK_I], thereby allowing it to throttle the cycle speed. Any number of wait states may be added.

CLOCK EDGE 1: SLAVE latches data on [DAT_O()].
MASTER latches [TAGN_I].
MASTER negates [STB_O] and [CYC_O] to indicate the end of the cycle.
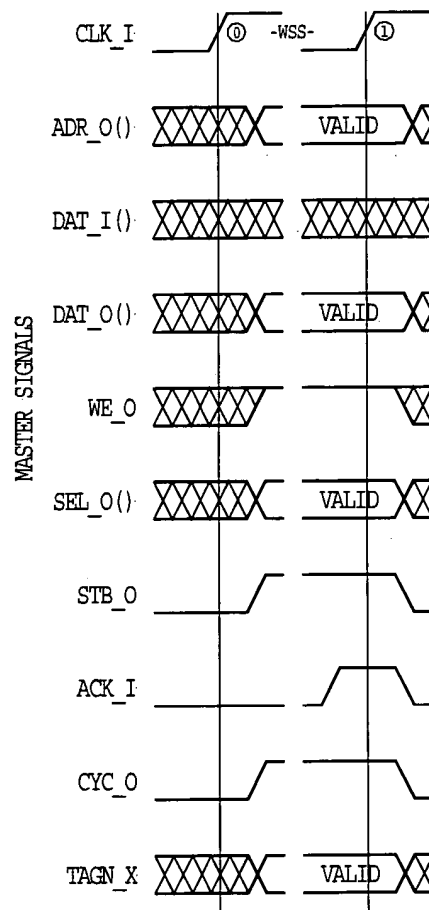
Figure 3-4. SINGLE WRITE cycle.

3.3 BLOCK READ / WRITE Cycles

The BLOCK transfer cycles perform multiple data transfers. They are very similar to single READ and WRITE cycles, but have a few special modifications to support multiple transfers.

During BLOCK cycles, the interface basically performs SINGLE READ/WRITE cycles as described above. However, the BLOCK cycles are modified somewhat so that these individual cycles are combined together to form a single BLOCK cycle. This function is most useful when multiple MASTERs are used on the interconnect. For example, if the SLAVE is a shared (dual port) memory, then an arbiter for that memory can determine when one MASTER is done with it so that another can gain access to the memory.

As shown in Figure 3-5, the [CYC_O] signal is asserted for the duration of a BLOCK cycle. This signal can be used to request permission to access from a shared resource from a local arbiter, and hold the access until the end of the current cycle. During each of the data transfers (within the block transfer), the normal handshaking protocol between [STB_O] and [ACK_I] is maintained.
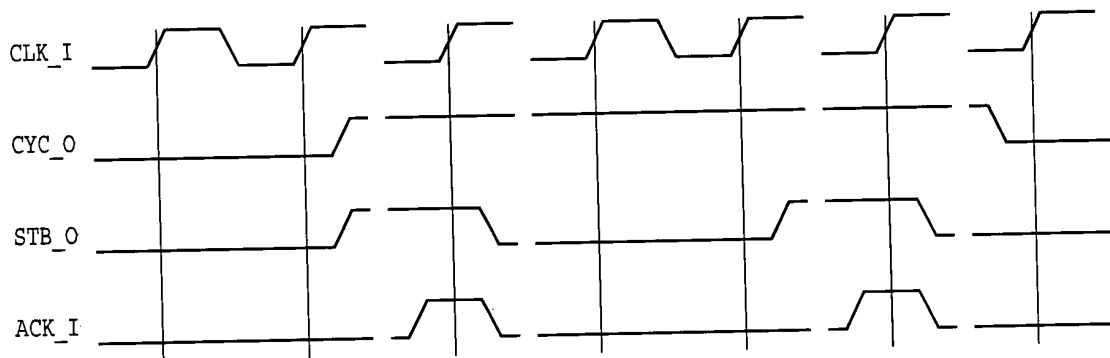
Figure 3-5. Use of [CYC_O] signal during BLOCK cycles.

It should be noted that the [CYC_O] signal does not necessarily rise and fall at the same time as [STB_O]. [CYC_O] may be asserted at the same time as [STB_O], or one or more [CLK_I] edges before [STB_O]. Similarly, [CYC_O] may be negated at the same time as [STB_O], or after an indeterminate number of [CLK_I] cycles.

RULE 3.70
All MASTER and SLAVE interfaces that support BLOCK cycles MUST conform to the timing requirements given in sections 3.3.1 and 3.3.2.

PERMISSION 3.25
MASTER and SLAVE interfaces MAY be designed so that they do not support the BLOCK cycles.

3.3.1 BLOCK READ Cycle

Figure 3-6 shows a BLOCK READ cycle. The BLOCK cycle is capable of a data transfer on every clock cycle. However, this example also shows how the MASTER and the SLAVE interfaces can both throttle the bus transfer rate by inserting wait states. A total of five transfers are shown. After the second transfer the MASTER inserts a wait state. After the fourth transfer the SLAVE inserts a wait state. The cycle is terminated after the fifth transfer. The protocol for this transfer works as follows:

CLOCK EDGE 0: MASTER presents [ADR_O()] and [TAGN_O].
MASTER negates [WE_O] to indicate a READ cycle.
MASTER presents bank select [SEL_O()] to indicate where it expects data.
MASTER asserts [CYC_O] to indicate the start of the cycle.
MASTER asserts [STB_O].

Note: the MASTER must assert [CYC_O] and/or [TAGN_O] at, or anytime before, clock edge 1. The use of [TAGN_O] is optional.

SETUP, EDGE 1: SLAVE decodes inputs, and responds by asserting [ACK_I].
SLAVE presents valid data on [DAT_I].
SLAVE presents [TAGN_O].
MASTER monitors [TAGN_I].
MASTER monitors [ACK_I], and prepares to latch data on [DAT_I()].

CLOCK EDGE 1: MASTER latches data on [DAT_I()].
MASTER latches [TAGN_I].
MASTER presents new [ADR_O()] and [TAGN_O].
MASTER presents new bank select [SEL_O()] to indicate where data is.

SETUP, EDGE 2: SLAVE decodes inputs, and responds by asserting [ACK_I].
SLAVE presents valid data on [DAT_I].
SLAVE presents [TAGN_O].
MASTER monitors [TAGN_I].
MASTER monitors [ACK_I], and prepares to latch data on [DAT_I()].

CLOCK EDGE 2: MASTER latches data on [DAT_I()].
MASTER latches [TAGN_I].
MASTER negates [STB_O] to introduce a wait state (-WSM-).

SETUP, EDGE 3: SLAVE negates [ACK_I] in response to [STB_O].

Note: any number of wait states can be inserted by the MASTER.

CLOCK EDGE 3: MASTER presents new [ADR_O()] and [TAGN_O].
MASTER presents new bank select [SEL_O()].
MASTER asserts [STB_O].

SETUP, EDGE 4: SLAVE decodes inputs, and responds by asserting [ACK_I].
SLAVE presents valid data on [DAT_I].
SLAVE presents [TAGN_O].
MASTER monitors [TAGN_I].
MASTER monitors [ACK_I], and prepares to latch data on [DAT_I()].

CLOCK EDGE 4: MASTER latches data on [DAT_I()].
MASTER presents [ADR_O()] and [TAGN_O].
MASTER latches [TAGN_I].
MASTER presents new bank select [SEL_O()] to indicate where it expects data.

SETUP, EDGE 5: SLAVE decodes inputs, and responds by asserting [ACK_I].
SLAVE presents valid data on [DAT_I].
SLAVE presents [TAGN_O].
MASTER monitors [TAGN_I].
MASTER monitors [ACK_I], and prepares to latch data on [DAT_I()].

CLOCK EDGE 5: MASTER latches data on [DAT_I()].
MASTER latches [TAGN_I].
SLAVE negates [ACK_I] to introduce a wait state.

Note: any number of wait states can be inserted by the SLAVE at this point.

SETUP, EDGE 6: SLAVE decodes inputs, and responds by asserting [ACK_I].
SLAVE presents valid data on [DAT_I].
MASTER monitors [ACK_I], and prepares to latch data on [DAT_I()].

CLOCK EDGE 6: MASTER latches data on [DAT_I()].
MASTER terminates cycle by negating [STB_O] and [CYC_O].
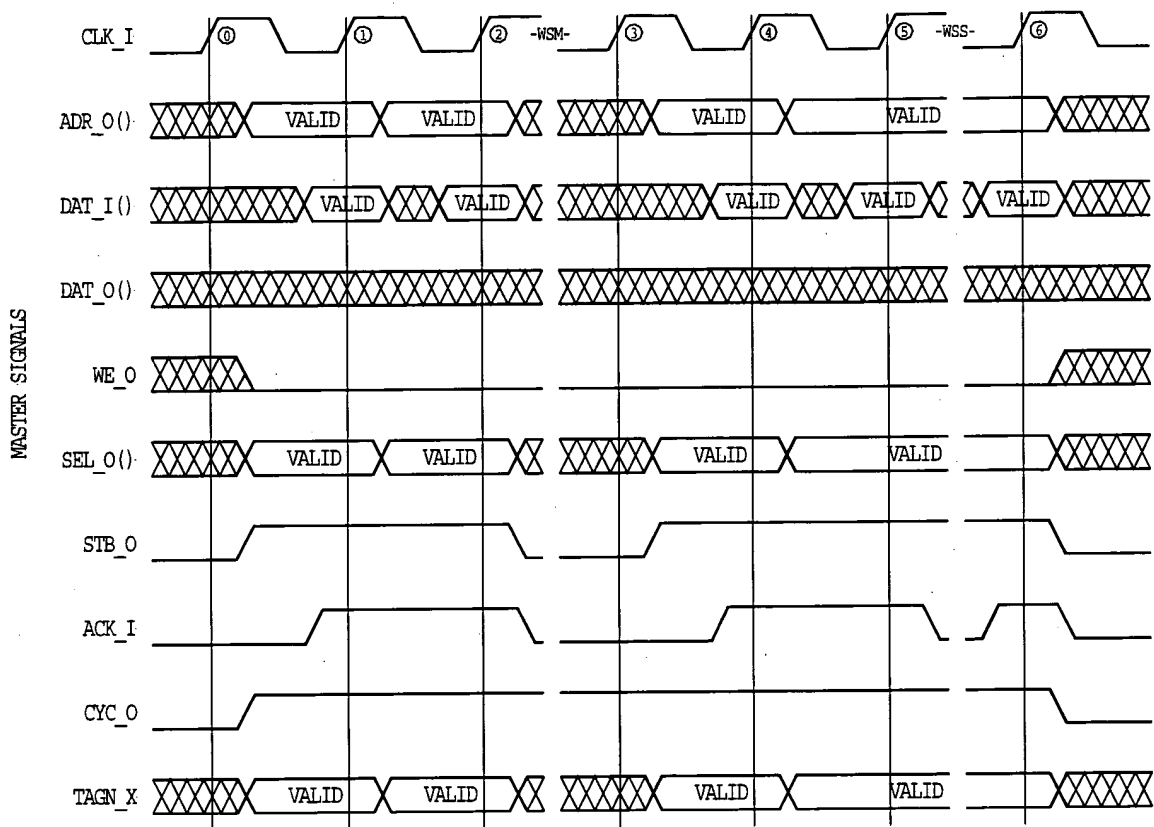
Figure 3-6. BLOCK READ cycle.

3.3.2 BLOCK WRITE Cycle

Figure 3-7 shows a BLOCK WRITE cycle. The BLOCK cycle is capable of a data transfer on every clock cycle. However, this example also shows how the MASTER and the SLAVE interfaces can both throttle the bus transfer rate by inserting wait states. A total of five transfers are shown. After the second transfer the MASTER inserts a wait state. After the fourth transfer the SLAVE inserts a wait state. The cycle is terminated after the fifth transfer. The protocol for this transfer works as follows:

CLOCK EDGE 0: MASTER presents [ADR_O()] and [TAGN_O].
MASTER asserts [WE_O] to indicate a WRITE cycle.
MASTER presents bank select [SEL_O()] to indicate where it expects data.
MASTER asserts [CYC_O] and [TAGN_O] to indicate cycle start.
MASTER asserts [STB_O].

Note: the MASTER must assert [CYC_O] and/or [TAGN_O] at, or anytime before, clock edge 1. The use of [TAGN_O] is optional.

SETUP, EDGE 1: SLAVE decodes inputs, and responds by asserting [ACK_I].
SLAVE prepares to latch data on [DAT_O].
SLAVE presents [TAGN_O].
MASTER monitors [TAGN_I].
MASTER monitors [ACK_I], and prepares to terminate current data phase.

CLOCK EDGE 1: SLAVE latches data on [DAT_O()].
MASTER latches [TAGN_I].
MASTER presents [ADR_O()] and [TAGN_O].
MASTER presents new bank select [SEL_O()].

SETUP, EDGE 2: SLAVE decodes inputs, and responds by asserting [ACK_I].
SLAVE prepares to latch data on [DAT_O].
SLAVE presents [TAGN_O].
MASTER monitors [TAGN_I].
MASTER monitors [ACK_I], and prepares to terminate current data phase.

CLOCK EDGE 2: SLAVE latches data on [DAT_O()].
MASTER latches [TAGN_I].
MASTER negates [STB_O] to introduce a wait state (-WSM-).

SETUP, EDGE 3: SLAVE negates [ACK_I] in response to [STB_O].

Note: any number of wait states can be inserted by the MASTER at this point.

CLOCK EDGE 3: MASTER presents [ADR_O()] and [TAGN_O].
MASTER presents bank select [SEL_O()] to indicate where it expects data.

MASTER asserts [STB_O].

SETUP, EDGE 4: SLAVE decodes inputs, and responds by asserting [ACK_I].
SLAVE prepares to latch data on [DAT_O].
SLAVE presents [TAGN_O].
MASTER monitors [TAGN_I].
MASTER monitors [ACK_I], and prepares to terminate data phase.

CLOCK EDGE 4: SLAVE latches data on [DAT_O()].
MASTER latches [TAGN_I].
MASTER presents [ADR_O()] and [TAGN_O].
MASTER presents new bank select [SEL_O()] to indicate where it expects data.

SETUP, EDGE 5: SLAVE decodes inputs, and responds by asserting [ACK_I].
SLAVE prepares to latch data on [DAT_O].
SLAVE presents [TAGN_O].
MASTER monitors [TAGN_I].
MASTER monitors [ACK_I], and prepares to terminate data phase.

CLOCK EDGE 5: SLAVE latches data on [DAT_O()].
SLAVE negates [ACK_I] to introduce a wait state.
MASTER latches [TAGN_I].

Note: any number of wait states can be inserted by the SLAVE at this point.

SETUP, EDGE 6: SLAVE decodes inputs, and responds by asserting [ACK_I].
SLAVE prepares to latch data on [DAT_O].
MASTER monitors [ACK_I], and prepares to terminate data phase.

CLOCK EDGE 6: SLAVE latches data on [DAT_O()].
MASTER terminates cycle by negating [STB_O] and [CYC_O].
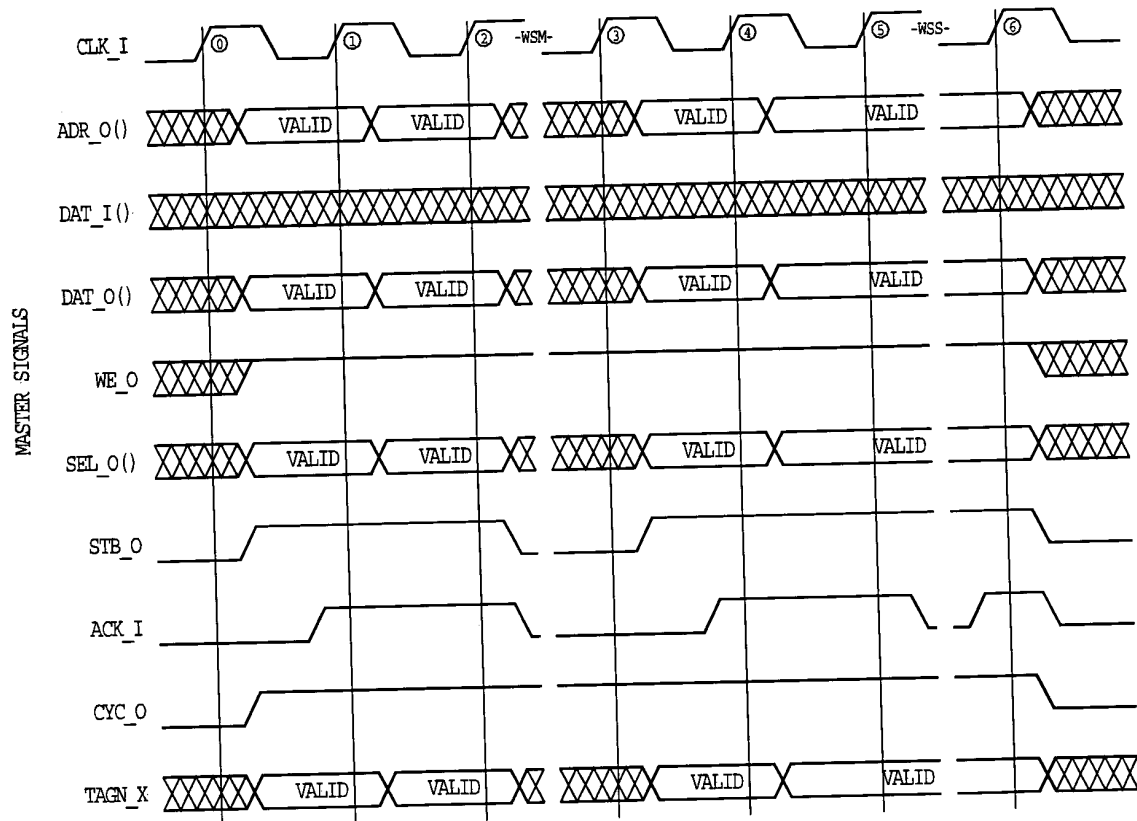
Figure 3-7. BLOCK WRITE cycle.

3.4 RMW Cycle

The RMW (read-modify-write) cycle is used for indivisible semaphore operations. During the first half of the cycle a single read data transfer is performed. During the second half of the cycle a write data transfer is performed. The [CYC_O] signal remains asserted during both halves of the cycle.

RULE 3.75
All MASTER and SLAVE interfaces that support RMW cycles MUST conform to the timing requirements given in section 3.4.

PERMISSION 3.30
MASTER and SLAVE interfaces MAY be designed so that they do not support the RMW cycles.

Figure 3-8 shows a read-modify-write (RMW) cycle. The RMW cycle is capable of a data transfer on every clock cycle. However, this example also shows how the MASTER and the SLAVE interfaces can both throttle the bus transfer rate by inserting wait states. Two transfers are shown. After the first (read) transfer, the MASTER inserts a wait state. During the second transfer the SLAVE inserts a wait state. The protocol for this transfer works as follows:

CLOCK EDGE 0: MASTER presents [ADR_O()] and [TAGN_O].
MASTER negates [WE_O] to indicate a READ cycle.
MASTER presents bank select [SEL_O()] to indicate where it expects data.
MASTER asserts [CYC_O] and [TAGN_O] to indicate the start of cycle.
MASTER asserts [STB_O].

Note: the MASTER must assert [CYC_O] and/or [TAGN_O] at, or anytime before, clock edge 1. The use of [TAGN_O] is optional.

SETUP, EDGE 1: SLAVE decodes inputs, and responds by asserting [ACK_I].
SLAVE presents valid data on [DAT_I].
SLAVE presents [TAGN_O].
MASTER monitors [TAGN_I].
MASTER monitors [ACK_I], and prepares to latch data on [DAT_I()].

CLOCK EDGE 1: MASTER latches data on [DAT_I()].
MASTER latches [TAGN_I].
MASTER negates [STB_O] to introduce a wait state (-WSM-).

SETUP, EDGE 2: SLAVE negates [ACK_I] in response to [STB_O].
MASTER asserts [WE_O] to indicate a WRITE cycle.

Note: any number of wait states can be inserted by the MASTER at this point.

CLOCK EDGE 2: MASTER presents the same [ADR_O()] and [TAGN_O] as was on clock 1.
                  MASTER presents WRITE data on [DAT_O()].
                  MASTER presents new bank select [SEL_O()].
                  MASTER asserts [STB_O].

SETUP, EDGE 3: SLAVE decodes inputs, and responds by asserting [ACK_I] (when ready).
                  SLAVE presents valid data on [DAT_I].
                  SLAVE presents [TAGN_O].
                  MASTER monitors [TAGN_I].
                  MASTER monitors [ACK_I], and prepares to latch data on [DAT_I()].

                  Note: any number of wait states can be inserted by the SLAVE at this point.

CLOCK EDGE 3: SLAVE latches data on [DAT_O()].
                  MASTER latches [TAGN_I].
                  MASTER negates [STB_O] and [CYC_O] indicating the end of the cycle.
                  SLAVE negates [ACK_I] in response to negated [STB_O].
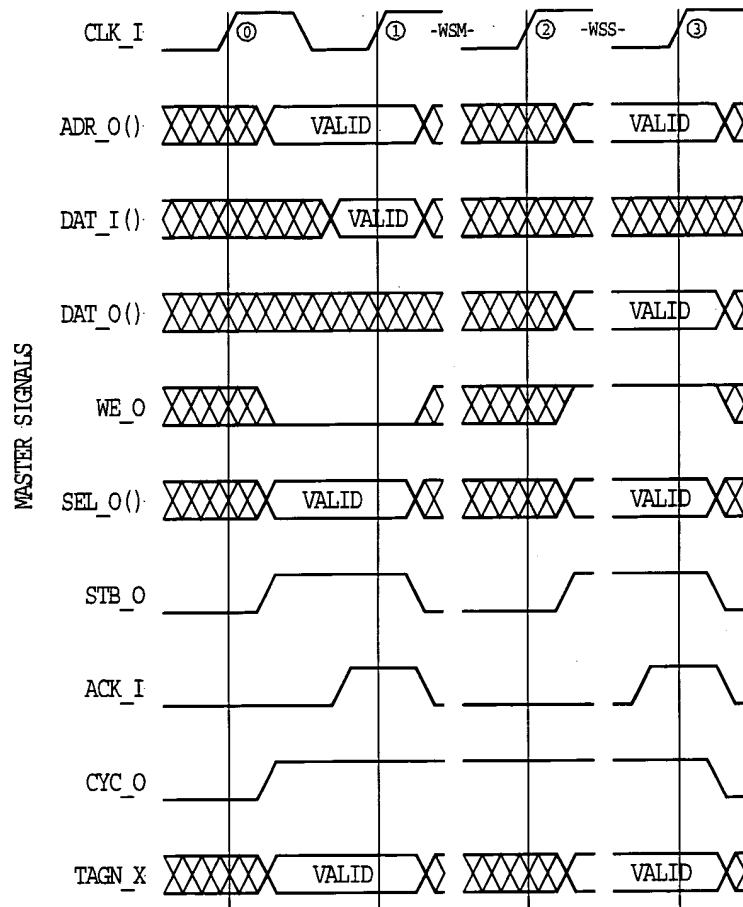
Figure 3-8. RMW cycle.

3.5 Data Organization

Data organization refers to the ordering of data during transfers. There are two general types of ordering which are called BIG ENDIAN and LITTLE ENDIAN. BIG ENDIAN refers to data ordering where the most significant portion of an operand is stored at the lower address. LITTLE ENDIAN refers to data ordering where the most significant portion of an operand is stored at the higher address. The WISHBONE architecture supports both methods of data ordering.

3.5.1 Nomenclature

A BYTE(N), WORD(N), DWORD(N) and QWORD(N) nomenclature is used to define data ordering. These terms are defined in Table 3-1. Figure 3-9 shows the operand locations for input and output data ports.

Table 3-1. Data Transfer Nomenclature		
Nomenclature	Granularity	Description
BYTE(N)	8-bit	An 8-bit BYTE transfer at address 'N'.
WORD(N)	16-bit	A 16-bit WORD transfer at address 'N'.
DWORD(N)	32-bit	A 32-bit Double WORD transfer at address 'N'.
QWORD(N)	64-bit	A 64-bit Quadruple WORD transfer at address 'N'.

The table also defines the granularity of the interface. This indicates the minimum unit of data transfer that is supported by the interface. For example, the smallest operand that can be passed through a port with 16-bit granularity is a 16-bit WORD. In this case, an 8-bit operand cannot be transferred.

Figure 3-10 shows an example of how the 64-bit value of 0x0123456789ABCDEF is transferred through BYTE, WORD, DWORD and QWORD ports using BIG ENDIAN data organization. Through the 64-bit QWORD port the number is directly transferred with the most significant bit at DAT_I / DAT_O(63). The least significant bit is at DAT_I / DAT_O(0). However, when the same operand is transferred through a 32-bit DWORD port, it is split into two bus cycles. The two bus cycles are each 32-bits in length, with the most significant DWORD transferred at the lower address, and the least significant DWORD transferred at the upper address. A similar situation applies to the WORD and BYTE cases.

Figure 3-11 shows an example of how the 64-bit value of 0x0123456789ABC is transferred through BYTE, WORD, DWORD and QWORD ports using LITTLE ENDIAN data organization. Through the 64-bit QWORD port the number is directly transferred with the most significant bit at DAT_I / DAT_O(63). The least significant bit is at DAT_I / DAT_O(0). However, when the same operand is transferred through a 32-bit DWORD port, it is split into two bus cycles. The two bus cycles are each 32-bits in length, with the least significant DWORD transferred at the lower address, and the most significant DWORD transferred at the upper address. A similar situation applies to the WORD and BYTE cases.

QWORD(0)							
DWORD(0)				DWORD(1)			
WORD(0)		WORD(1)		WORD(2)		WORD(3)	
BYTE(0)	BYTE(1)	BYTE(2)	BYTE(3)	BYTE(4)	BYTE(5)	BYTE(6)	BYTE(7)

63   DAT_I / DAT_O   00

(a) BIG ENDIAN BYTE, WORD, DWORD and QWORD positioning in a 64-bit operand.

63   DAT_I / DAT_O   00

QWORD(0)							
DWORD(1)				DWORD(0)			
WORD(3)		WORD(2)		WORD(1)		WORD(0)	
BYTE(7)	BYTE(6)	BYTE(5)	BYTE(4)	BYTE(3)	BYTE(2)	BYTE(1)	BYTE(0)
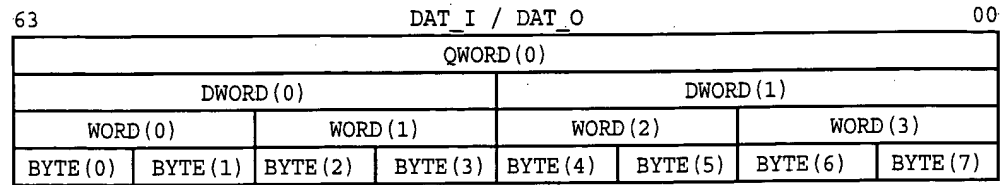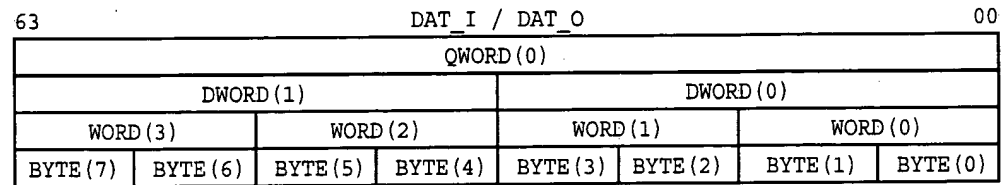
(b) LITTLE ENDIAN BYTE, WORD, DWORD and QWORD positioning in a 64-bit operand.

63   DAT_I / DAT_O   00

QWORD(0)

QWORD
ORDERING

DAT_I / DAT_O
07   00

OFFSET ADDRESS	BYTE ORDERING
7	BYTE(7)
6	BYTE(6)
5	BYTE(5)
4	BYTE(4)
3	BYTE(3)
2	BYTE(2)
1	BYTE(1)
0	BYTE(0)

15 DAT_I / DAT_O 00

WORD(3)
WORD(2)
WORD(1)
WORD(0)

WORD
ORDERING

31   DAT_I / DAT_O   00

DWORD(1)
DWORD(0)

DWORD
ORDERING
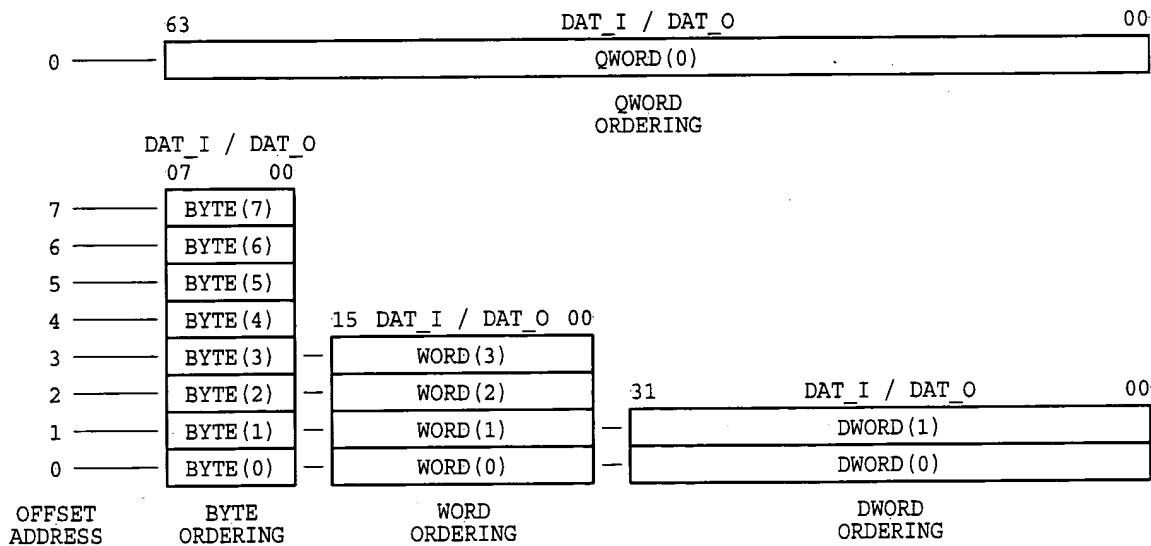
(c) Address nomenclature.

Figure 3-9. Operand locations for input and output data ports.
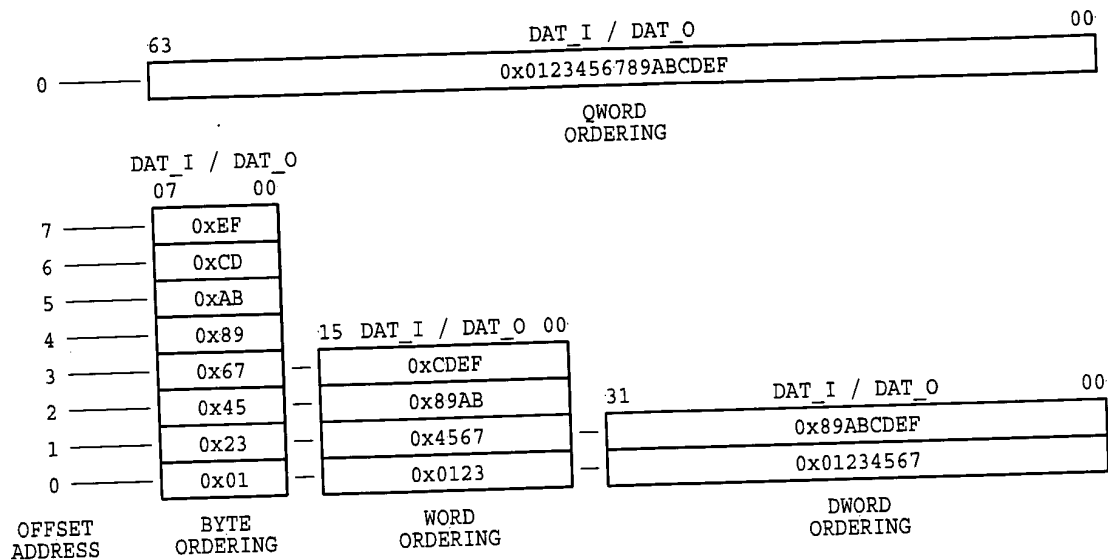
Figure 3-10. Example showing a variety of BIG ENDIAN transfers over various port sizes.
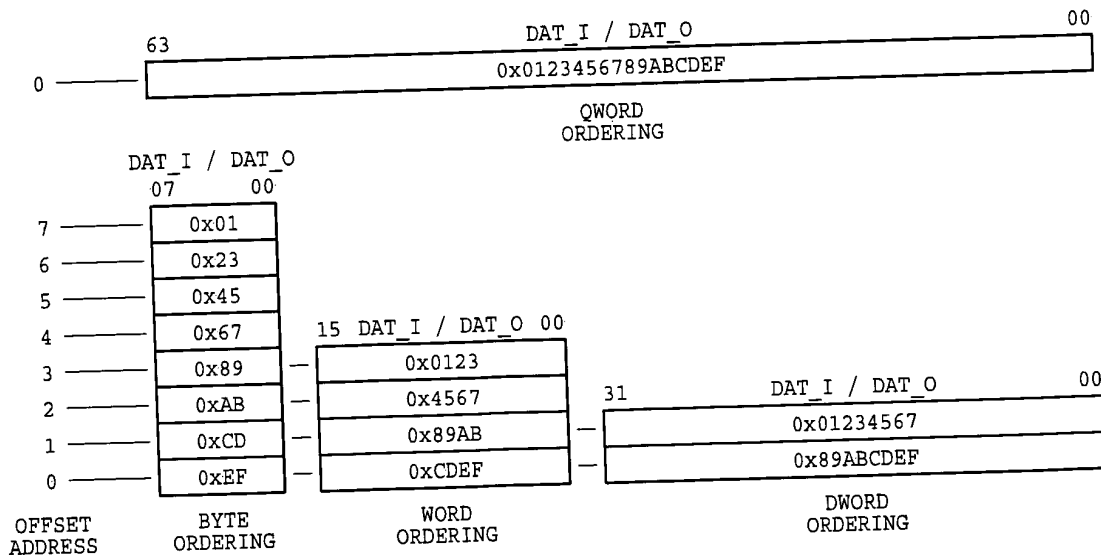
Figure 3-11. Example showing a variety of LITTLE ENDIAN transfers over various port sizes.

RULE 3.80
Data organization MUST conform to the ordering indicated in Figure 3-9.

3.5.2 Transfer Sequencing

The sequence in which data is transferred through a port is not regulated by this specification. For example, a 64-bit operand through a 32-bit port will take two bus cycles. However, the specification does not require that the lower or upper DWORD be transferred first.

RECOMMENDATION 3.10
Design interfaces so that data is transferred sequentially from lower addresses to a higher addresses.

OBSERVATION 3.30
The sequence in which an operand is transferred through a data port is not highly regulated by the specification. That is because different IP cores may produce the data in different ways. The sequence is therefore application-specific.

3.5.3 Data Organization for 64-bit Ports

RULE 3.85
Data organization on 64-bit ports MUST conform to Figure 3-12.

64-bit Data Bus With 8-bit (BYTE) Granularity									
	Address Range:	Active Portion of Data Bus							
	ADR_I ADR_O (63..03)	DAT_I DAT_O (63..56)	DAT_I DAT_O (55..48)	DAT_I DAT_O (47..40)	DAT_I DAT_O (39..32)	DAT_I DAT_O (31..24)	DAT_I DAT_O (23..16)	DAT_I DAT_O (15..08)	DAT_I DAT_O (07..00)
	Active Select Line	SEL_I(7) SEL_O(7)	SEL_I(6) SEL_O(6)	SEL_I(5) SEL_O(5)	SEL_I(4) SEL_O(4)	SEL_I(3) SEL_O(3)	SEL_I(2) SEL_O(2)	SEL_I(1) SEL_O(1)	SEL_I(0) SEL_O(0)
BYTE Ordering	BIG ENDIAN	BYTE(0)	BYTE(1)	BYTE(2)	BYTE(3)	BYTE(4)	BYTE(5)	BYTE(6)	BYTE(7)
	LITTLE ENDIAN	BYTE(7)	BYTE(6)	BYTE(5)	BYTE(4)	BYTE(3)	BYTE(2)	BYTE(1)	BYTE(0)

64-bit Data Bus With 16-bit (WORD) Granularity					
	Address Range	Active Portion of Data Bus			
	ADR_I ADR_O (63..02)	DAT_I DAT_O (63..48)	DAT_I DAT_O (47..32)	DAT_I DAT_O (31..16)	DAT_I DAT_O (15..00)
	Active Select Line	SEL_I(3) SEL_O(3)	SEL_I(2) SEL_O(2)	SEL_I(1) SEL_O(1)	SEL_I(0) SEL_O(0)
WORD Ordering	BIG ENDIAN	WORD(0)	WORD(1)	WORD(2)	WORD(3)
	LITTLE ENDIAN	WORD(3)	WORD(2)	WORD(1)	WORD(0)

64-bit Data Bus With 32-bit (DWORD) Granularity			
	Address Range	Active Portion of Data Bus	
	ADR_I ADR_O (63..01)	DAT_I DAT_O (63..32)	DAT_I DAT_O (31..00)
	Active Select Line	SEL_I(1) SEL_O(1)	SEL_I(0) SEL_O(0)
DWORD Ordering	BIG ENDIAN	DWORD(0)	DWORD(1)
	LITTLE ENDIAN	DWORD(1)	DWORD(0)

64-bit Data Bus With 64-bit (QWORD) Granularity		
	Address Range	Active Portion of Data Bus
	ADR_I ADR_O (63..00)	DAT_I DAT_O (63..00)
	Active Select Line	SEL_I(0) SEL_O(0)
QWORD Ordering	BIG ENDIAN	QWORD(0)
	LITTLE ENDIAN	QWORD(0)

Figure 3-12. Data organization for 64-bit ports.

3.5.4 Data Organization for 32-bit Ports

RULE 3.90
Data organization on 32-bit ports MUST conform to Figure 3-13.

32-bit Data Bus With 8-bit (BYTE) Granularity					
	Address Range	Active Portion of Data Bus			
	ADR_I ADR_O (63..02)	DAT_I DAT_O (31..24)	DAT_I DAT_O (23..16)	DAT_I DAT_O (15..08)	DAT_I DAT_O (07..00)
	Active Select Line	SEL_I(3) SEL_O(3)	SEL_I(2) SEL_O(2)	SEL_I(1) SEL_O(1)	SEL_I(0) SEL_O(0)
BYTE Ordering	BIG ENDIAN	BYTE(0) BYTE(4)	BYTE(1) BYTE(5)	BYTE(2) BYTE(6)	BYTE(3) BYTE(7)
	LITTLE ENDIAN	BYTE(3) BYTE(7)	BYTE(2) BYTE(6)	BYTE(1) BYTE(5)	BYTE(0) BYTE(4)

32-bit Data Bus With 16-bit (WORD) Granularity			
	Address Range	Active Portion of Data Bus	
	ADR_I ADR_O (63..01)	DAT_I DAT_O (31..16)	DAT_I DAT_O (15..00)
	Active Select Line	SEL_I(1) SEL_O(1)	SEL_I(0) SEL_O(0)
WORD Ordering	BIG ENDIAN	WORD(0) WORD(2)	WORD(1) WORD(3)
	LITTLE ENDIAN	WORD(1) WORD(3)	WORD(0) WORD(2)

32-bit Data Bus With 32-bit (DWORD) Granularity		
	Address Range	Active Portion of Data Bus
	ADR_I ADR_O (63..00)	DAT_I DAT_O (31..00)
	Active Select Line	SEL_I(0) SEL_O(0)
DWORD Ordering	BIG ENDIAN	DWORD(0) DWORD(1)
	LITTLE ENDIAN	DWORD(0) DWORD(1)

Figure 3-13. Data organization for 32-bit ports.

3.5.5 Data Organization for 16-bit Ports

RULE 3.95
Data organization on 16-bit ports MUST conform to Figure 3-14.

16-bit Data Bus With 8-bit (BYTE) Granularity			
	Address Range	Active Portion of Data Bus	
	ADR_I ADR_O (63..01)	DAT_I DAT_O (15..08)	DAT_I DAT_O (07..00)
	Active Select Line	SEL_I(1) SEL_O(1)	SEL_I(0) SEL_O(0)
BYTE Ordering	BIG ENDIAN	BYTE(0) BYTE(2) BYTE(4) BYTE(6)	BYTE(1) BYTE(3) BYTE(5) BYTE(7)
	LITTLE ENDIAN	BYTE(1) BYTE(3) BYTE(5) BYTE(7)	BYTE(0) BYTE(2) BYTE(4) BYTE(6)

16-bit Data Bus With 16-bit (WORD) Granularity		
	Address Range	Active Portion of Data Bus
	ADR_I ADR_O (63..00)	DAT_I DAT_O (15..00)
	Active Select Line	SEL_I(0) SEL_O(0)
WORD Ordering	BIG ENDIAN	WORD(0) WORD(1) WORD(2) WORD(3)
	LITTLE ENDIAN	WORD(0) WORD(1) WORD(2) WORD(3)

Figure 3-14. Data organization for 16-bit ports.

3.5.6 Data Organization for 8-bit Ports

RULE 3.100

Data organization on 8-bit ports MUST conform to Figure 3-15.

8-bit Data Bus With 8-bit (BYTE) Granularity		
	Address Range	Active Portion of Data Bus
	ADR_I ADR_O (63..00)	DAT_I DAT_O (07..00)
	Active Select Line	SEL_I(0) SEL_O(0)
BYTE Ordering	BIG ENDIAN	BYTE(0) BYTE(1) BYTE(2) BYTE(3) BYTE(4) BYTE(5) BYTE(6) BYTE(7)
	LITTLE ENDIAN	BYTE(0) BYTE(1) BYTE(2) BYTE(3) BYTE(4) BYTE(5) BYTE(6) BYTE(7)

Figure 3-15. Data organization for 8-bit ports.

3.6 References

Cohen, Danny. *On Holy Wars and a Plea for Peace*. IEEE Computer Magazine, October 1981. Pages 49-54. [Description of BIG ENDIAN and LITTLE ENDIAN.]

Chapter 4 – Timing Specification

The WISHBONE specification is designed to provide the end user with very simple timing constraints. Although the application specific circuit(s) will vary in this regard, the interface itself is designed to work without the need for detailed timing specifications. In all cases, the only timing information that is needed by the end user is the maximum clock frequency (for [CLK_I]) that is passed to a place & route tool. The maximum clock frequency is dictated by the time delay between a positive clock edge on [CLK_I] to the setup on a stage further down the logical signal path. This delay is shown graphically in Figure 4-1, and is defined as Tpd,clk-su.
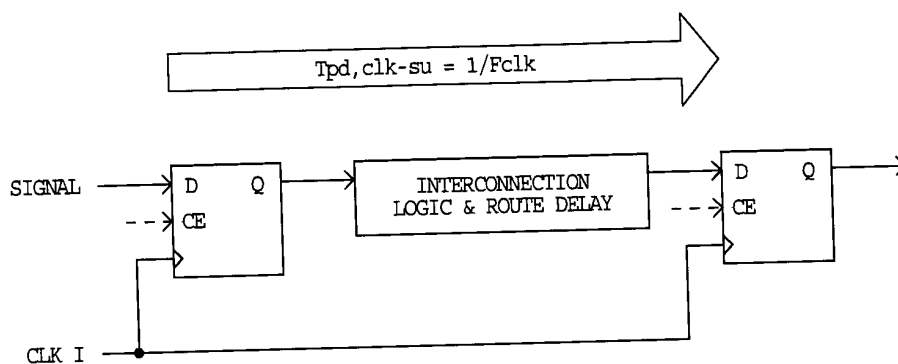
Figure 4-1. Definition for Tpd,clk-su.

RULE 4.00
The clock input [CLK_I] to each IP core MUST coordinate all activities for the internal logic within the WISHBONE interface. All WISHBONE output signals are registered at the rising edge of [CLK_I]. All WISHBONE input signals must be stable before the rising edge of [CLK_I].

PERMISSION 4.00
The user's place and route tool MAY be used to enforce this rule.

OBSERVATION 4.00
Most place and route tools can be easily configured to enforce this rule. Generally, it only requires a single timing specification for Tpd,clk-su.

RULE 4.05
The WISHBONE interface MUST use synchronous, RTL design methodologies that, given nearly infinitely fast gate delays, will operate over a nearly infinite range of clock frequencies on [CLK_I].

OBSERVATION 4.05
Realistically, the WISHBONE interface will never be expected to operate over a nearly infinite frequency range. However this requirement eliminates the need for non-portable timing constraints (that may work only on certain target devices).

OBSERVATION 4.10
The WISHBONE interface logic assumes that a low-skew clock distribution scheme is used on the target device, and that the clock-skew shall be low enough to permit reliable operation over the environmental conditions.

PERMISSION 4.05
The IP core connected to a WISHBONE interface MAY include application specific timing requirements.

RULE 4.10
The clock input [CLK_I] MUST have a duty cycle that is no less than 40%, and no greater than 60%.

PERMISSION 4.10
The SYSCON interface MAY use a variable clock generator. In these cases the clock frequency can be changed by the SYSCON interface so long as the clock edges remain clean and monotonic, and if the clock does not violate the duty cycle requirements.

PERMISSION 4.15
The SYSCON interface MAY use a gated clock generator. In these cases the clock shall be stopped in the low logic state. When the gated clock is stopped and started the clock edges must remain clean and monotonic.

SUGGESTION 4.00
When using a gated clock generator, turn the clock off when the WISHBONE interconnection is not busy. One way of doing this is to create a MASTER interface whose sole purpose is to acquire the WISHBONE interconnection and turn the clock off. This assures that the WISHBONE interconnection is not busy when gating the clock off. When the clock signal is restored the MASTER then releases the WISHBONE interconnection.

OBSERVATION 4.15
This specification does not attempt to govern the design of gated or variable clock generators.

SUGGESTION 4.10

Design an IP core so that all of the circuits (including the WISHBONE interconnect) follow the aforementioned RULEs, as this will make the core portable across a wide range of target devices and technologies.

References

Orton et al. <u>CHANGING CLOCK FREQUENCY</u> US Patent No. 6,118,306

Appendix A – WISHBONE Tutorial[4]

By: Wade D. Peterson, Silicore Corporation

The WISHBONE System-on-Chip (SoC) interconnection is a method for connecting digital circuits together to form an integrated circuit 'chip'. This tutorial provides an introduction to the WISHBONE design philosophy and its practical applications.

The WISHBONE architecture solves a very basic problem in integrated circuit design. That is, how to connect circuit functions together in a way that is simple, flexible and portable. The circuit functions are generally provided as 'IP Cores' (Intellectual Property Cores), which system integrators can purchase or make themselves.

Under this topology, IP Cores are the functional building blocks in the system. They are available in a wide variety of functions such as microprocessors, serial ports, disk interfaces, network controllers and so forth. Generally, the IP cores are developed independently from each other and are tied together and tested by a third party system integrator. WISHBONE aides the system integrator by standardizing the IP Core interfaces. This makes it much easier to connect the cores, and therefore much easier to create a custom System-on-Chip.

A.1 An Introduction to WISHBONE

WISHBONE uses a MASTER/SLAVE architecture. That means that functional modules with MASTER interfaces initiate data transactions to participating SLAVE interfaces. As shown in Figure A-1, the MASTERs and SLAVEs communicate through an interconnection interface called the INTERCON. The INTERCON is best thought of as a 'cloud' that contains circuits. These circuits allow MASTERs to communicate with SLAVEs.

The term 'cloud' is borrowed from the telecom community. Oftentimes, telephone systems are modeled as clouds that represent a system of telephone switches and transmission devices. Telephone handsets are connected to the cloud, and are used to make phone calls. The cloud itself represents a network that carries a telephone call from one location to another. The activity going on inside the cloud depends upon where the call is made and where it is going. For example, if a call is made to another office down the hall, then the cloud may represent a local telephone switch located in the same building. However, if the call is made across an ocean, then the cloud may represent a combination of copper, fiber optic and satellite transmission systems.

---

[4] This tutorial is not part of the WISHBONE specification.
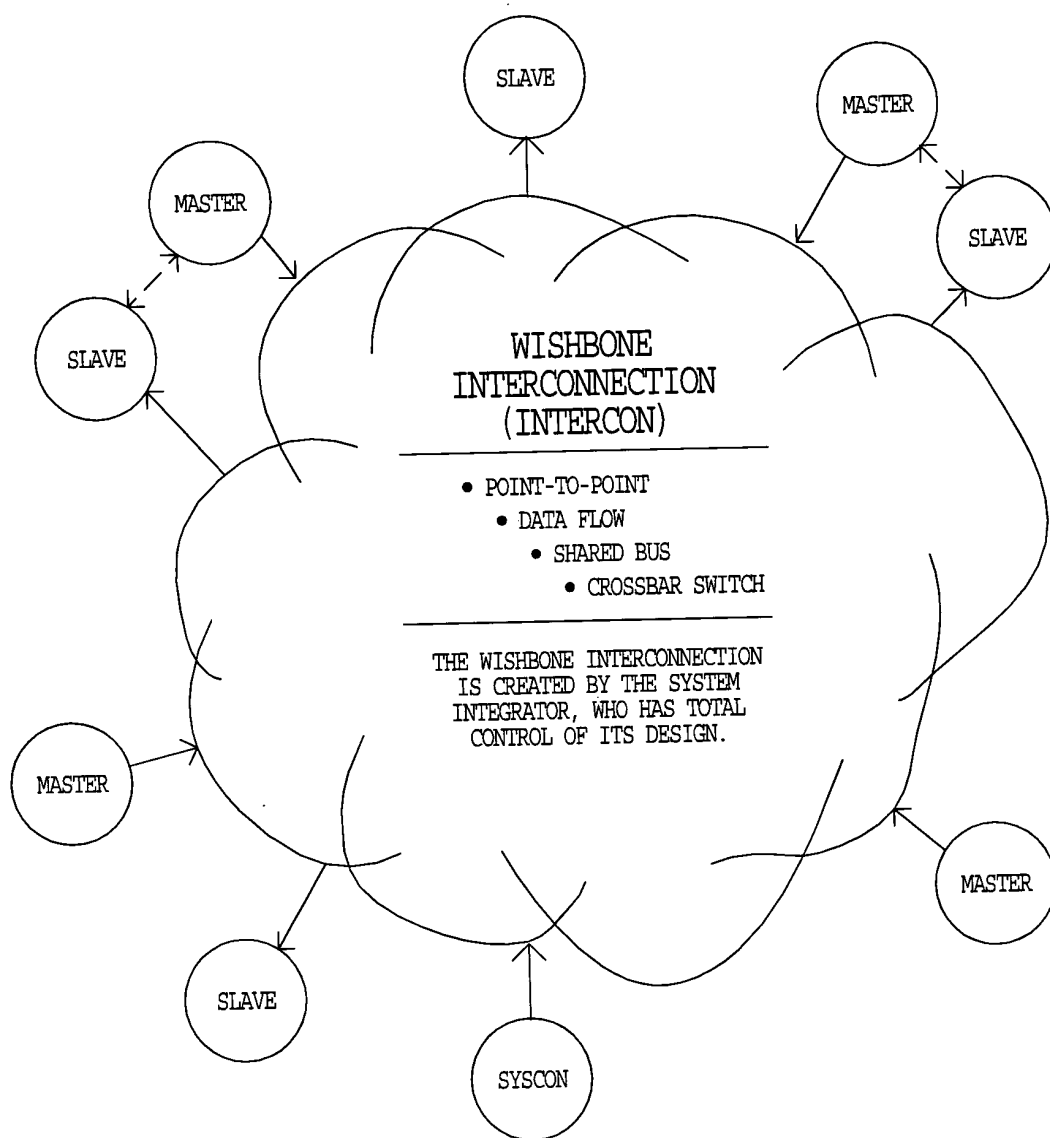
Figure A-1. The WISHBONE interconnection.

The cloud analogy is used because WISHBONE can be modeled in a similar way. MASTER and SLAVE interfaces (which are analogous to the telephones) communicate thorough an interconnection (which is analogous to the telephone network 'cloud'). The WISHBONE interconnection network can be changed by the system integrator to suit his or her own needs. In WISHBONE terminology this is called a *variable interconnection.*

Variable interconnection allows the system integrator to change the way in which the MASTER and SLAVE interfaces communicate with each other. For example, a pair of MASTER and SLAVE interfaces can communicate with point-to-point, data flow, shared bus or crossbar switch topologies.

The variable interconnection scheme is very different from that used in traditional microcomputer buses such as PCI, cPCI, VMEbus and ISA bus. Those systems use printed circuit backplanes with hardwired connectors. The interfaces on those buses can't be changed, which severely limits how microcomputer boards communicate with each other. WISHBONE overcomes this limitation by allowing the system integrator to change the system interconnection.

This is possible because integrated circuit chips have interconnection paths that can be adjusted. These are very flexible, and take the form of logic gates and routing paths. These can be 'programmed' into the chip using a variety of tools. For example, if the interconnection is described with a hardware description language like VHDL or Verilog®, then the system integrator has the ability to define and adjust the interconnection. Interconnection libraries can also be formed and shared.

The WISHBONE interconnection itself is nothing more than a large, synchronous circuit. It must be designed to *logically* operate over a nearly infinite frequency range. However, every integrated circuit has physical characteristics that limit the maximum frequency of the circuit. In WISHBONE terminology this is called a *variable timing specification.* This means that a WISHBONE compatible circuit will theoretically function normally at any speed, but that it's maximum speed will always be limited by the process technology of the integrated circuit.

At Silicore Corporation we generally define our WISHBONE interconnections using the VHDL hardware description language. These take the form of an ASCII file that contains the VHDL language instructions. This allows us to fully define our interconnections in a way that best fits the application. However, it also allows us to share our interconnections with others, who can adjust them to meet their own needs. It's important to note, though, that there's nothing magic about the interconnection itself. It's just a file, written with off-the-shelf tools, that fully describes the hardware in the interconnection.

A.2 Types of WISHBONE Interconnection

The WISHBONE variable interconnection allows the system integrator to change the way that IP cores connect to each other. There are four defined types of WISHBONE interconnection, and include:

- Point-to-point
- Data flow
- Shared bus
- Crossbar switch

A fifth possible type is the off-chip interconnection. However, off-chip implementations generally fit one of the other four basic types. For example, WISHBONE interfaces on two different integrated circuits can be connected using a point-to-point interconnection.

The WISHBONE specification does not dictate how any of these are implemented by the system integrator. That's because the interconnection itself is a type of IP Core interface called the INTERCON. The system integrator can use or modify an off-the-shelf INTERCON, or create their own.

A.2.1 Point-to-point Interconnection

The point-to-point interconnection is the simplest way to connect two WISHBONE IP cores together. As shown in Figure A-2, the point-to-point interconnection allows a single MASTER interface to connect to a single SLAVE interface. For example, the MASTER interface could be on a microprocessor IP core, and the SLAVE interface could be on a serial I/O port.
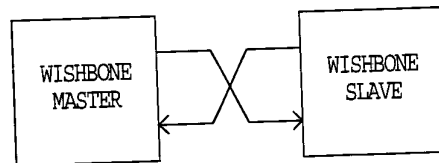
Figure A-2. The point-to-point interconnection.

A.2.2 Data Flow Interconnection

The data flow interconnection is used when data is processed in a sequential manner. As shown in Figure A-3, each IP core in the data flow architecture has both a MASTER and a SLAVE interface. Data flows from core-to-core. Sometimes this process is called *pipelining*.
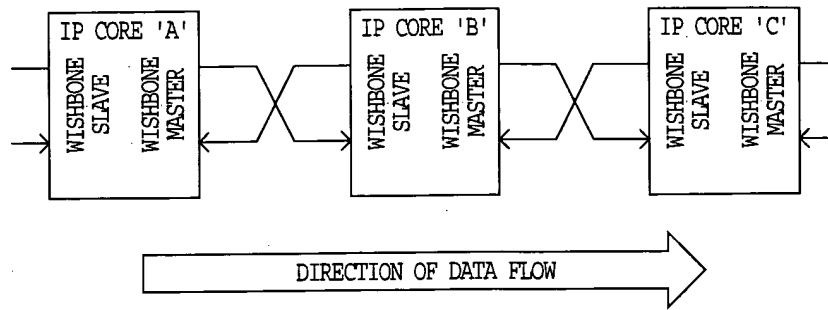
Figure A-3. The data flow interconnection.

The data flow architecture exploits parallelism, thereby speeding up execution time. For example, if each of the IP cores in the Figure represents a floating point processor, then the system has three times the number crunching potential of a single unit. This assumes, of course, that each IP Core takes an equal amount of time to solve its problem, and that the problem can be solved in a sequential manner. In actual practice this may or may not be true, but it does illustrate how the data flow architecture can provide a high degree of parallelism when solving problems.

A.2.3 Shared Bus Interconnection

The shared bus interconnection is useful for connecting two or more MASTERs with one or more SLAVEs. A block diagram is shown in Figure A-4. In this topology a MASTER initiates a bus cycle to a target SLAVE. The target SLAVE then participates in one or more bus cycles with the MASTER.

An arbiter (not shown in the Figure) determines when a MASTER may gain access to the shared bus. The arbiter acts like a 'traffic cop' to determine when and how each MASTER accesses the shared resource. Also, the type of arbiter is completely defined by the system integrator. For example, the shared bus can use a priority or a round robin arbiter. These grant the shared bus on a priority or equal basis, respectively.

The main advantage to this technique is that shared interconnection systems are relatively compact. Generally, it requires fewer logic gates and routing resources than other configurations, especially the crossbar switch. Its main disadvantage is that MASTERs may have to wait before gaining access to the interconnection. This degrades the overall speed at which a MASTER may transfer data.
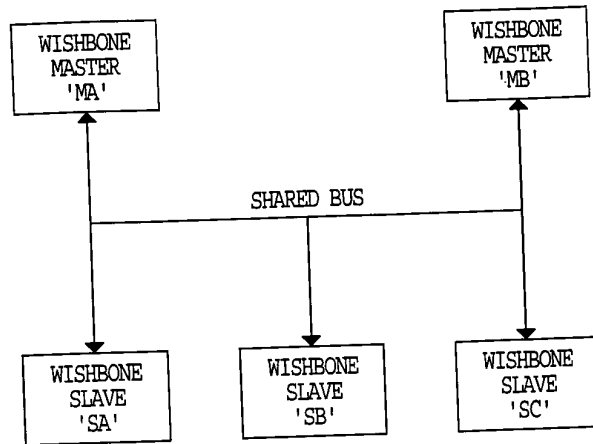
Figure A-4. Shared bus interconnection.

The WISHBONE specification does not dictate how the shared bus is implemented. Later on, we'll see that it can be made either with multiplexer or three-state buses. This gives the system integrator additional flexibility, as some logic chips work better with multiplexor logic, and some work better with three-state buses.

The shared bus is typically found in standard buses like PCI and VMEbus. There, a MASTER interface arbitrates for the common shared bus, and then communicates with a SLAVE. In both cases this is done with three-state buses.

A.2.4 Crossbar Switch Interconnection

The crossbar switch interconnection is used when connecting two or more WISHBONE MAS-TERs together so that each can access two or more SLAVEs. A block diagram is shown in Figure A-5. In the crossbar interconnection, a MASTER initiates an addressable bus cycle to a target SLAVE. An arbiter (not shown in the diagram) determines when each MASTER may gain access to the indicated SLAVE. Unlike the shared bus interconnection, the crossbar switch allows more than one MASTER to use the interconnection (as long as two MASTERs don't access the same SLAVE at the same time).
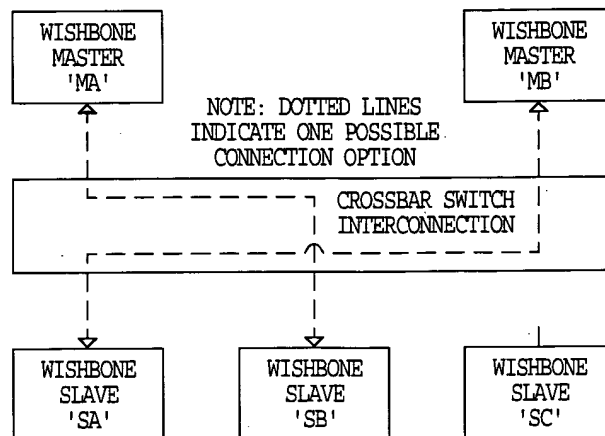
Figure A-5. Crossbar switch interconnection.

Under this method, each master arbitrates for a 'channel' on the switch. Once this is established, data is transferred between the MASTER and the SLAVE over a private communication link. The Figure shows two possible channels that may appear on the switch. The first connects MASTER 'MA' to SLAVE 'SB'. The second connects MASTER 'MB' to SLAVE 'SA'.

The overall data transfer rate of the crossbar switch is higher than shared bus mechanisms. For example, the figure shows two MASTER/SLAVE pairs interconnected at the same time. If each communication channel supports a data rate of 100 Mbyte/sec, then the two data pairs would operate in parallel at 200 Mbyte/sec. This scheme can be expanded to support extremely high data transfer rates.

One disadvantage of the crossbar switch is that it requires more interconnection logic and routing resources than shared bus systems. As a rule-of-thumb, a crossbar switch with two MASTERs and two SLAVEs takes twice as much interconnection logic as a similar shared bus system (with two MASTERs and two SLAVEs).

The crossbar interconnection is a typical configuration that one might find in microcomputer buses like[5] RACEway, SKY Channel or Myrinet.

---

[5] Raceway: ANSI/VITA 5-1994. SKYchannel: ANSI/VITA 10-1995. Myrinet: ANSI/VITA 26-1998. For more information about these standards see www.vita.com.

A.3 The WISHBONE Interface Signals

WISHBONE MASTER and SLAVE interfaces can be connected together in a number of ways. This requires that WISHBONE interface signals and bus cycles be designed in a very flexible and reusable manner. The signals were defined with the following requirements:

• The signals allow MASTER and SLAVE interfaces to support point-to-point, data flow, shared bus and crossbar switch interconnections.

• The signals allow three basic types of bus cycle. These include SINGLE READ/WRITE, BLOCK READ/WRITE and RMW (read-modify-write) bus cycles. The operation of these bus cycles are described below.

• A handshaking mechanism is used so that either the MASTER or the participating SLAVE interface can adjust the data transfer rate during a bus cycle. This allows the speed of each bus cycle (or phase) to be adjusted by either the MASTER or the SLAVE interface. This means that all WISHBONE bus cycles run at the speed of the slowest interface.

• The handshaking mechanism allows a participating SLAVE to accept a data transfer, reject a data transfer with an error or ask the MASTER to retry a bus cycle. The SLAVE does this by generating the [ACK_O], [ERR_O] or [RTY_O] signals respectively. Every interface must support the [ACK_O] signal, but the error and retry acknowledgement signals are optional.

• All signals on MASTER and SLAVE interfaces are either inputs or outputs, but are never bi-directional (i.e. three-state). This is because some FPGA and ASIC devices do not support bi-directional signals. However, it is permissible (and sometimes advantageous) to use bi-directional signals in the interconnection logic if the target device supports it.

• Address and data bus widths can be altered to fit the application. 8, 16, 32 and 64-bit data buses, and 0-64 bit address buses are defined.

• As shown in Figure A-6, all signals are arranged so that MASTER, SLAVE and SYSCON interfaces can be connected directly together to form a simple point-to-point interface. This allows very compact and efficient WISHBONE interfaces to be built. For example, WISHBONE could be used as the external system bus on a microprocessor IP Core. However, it's efficient enough so that it can be used for internal buses inside of the microprocessor.

• User defined signals in the form of 'tags' are allowed. This allows the system integrator to add special purpose signals to each WISHBONE interface. For example, the system integrator could add a parity bit to the address or data buses.

A comprehensive list of the WISHBONE signals and their descriptions is given in the specification.
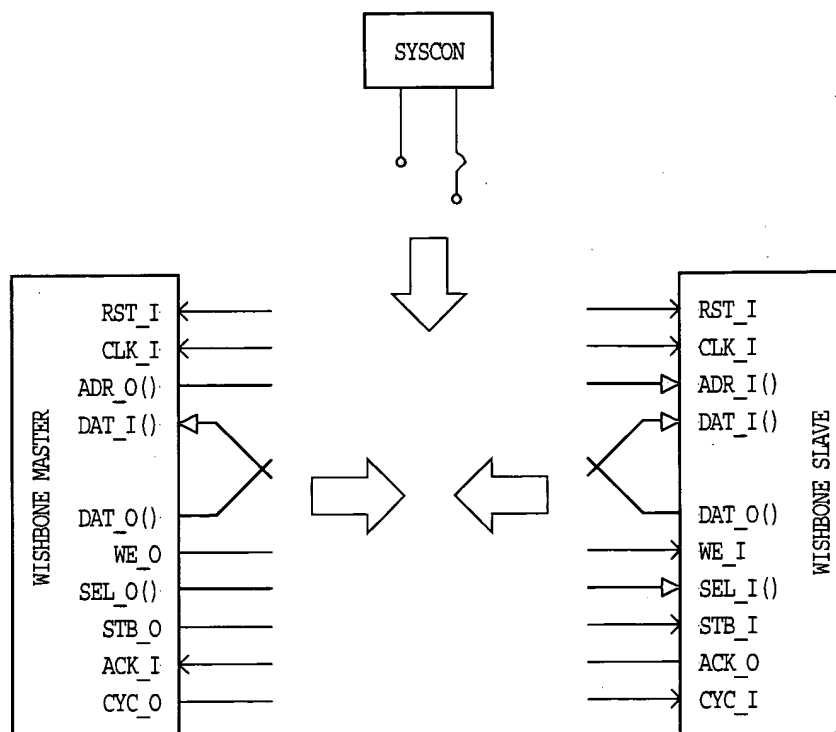
(A) FORMING A POINT-TO-POINT INTERCONNECTION.
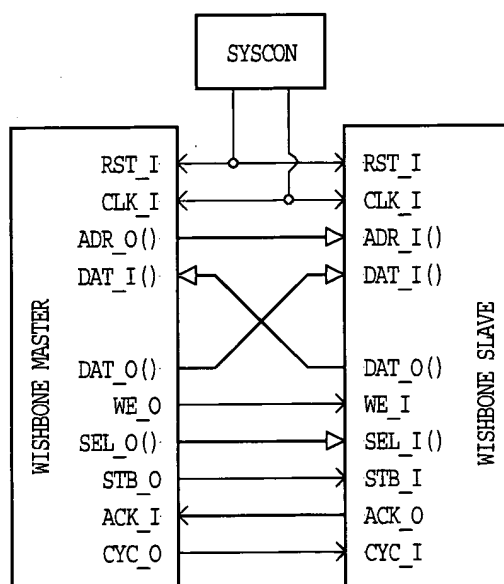
(B) POINT-TO-POINT INTERCONNECTION.

Figure A-6. The WISHBONE signals are selected to permit a MASTER, SLAVE and SYSCON interface to be directly connected, thereby forming a simple point-to-point interface.

A.4 The WISHBONE Bus Cycles

There are three types of defined WISHBONE bus cycles. They include:

- SINGLE READ/WRITE
- BLOCK READ/WRITE
- READ MODIFY WRITE (RMW)

A.4.1 SINGLE READ/WRITE Cycle

The SINGLE READ/WRITE is the most basic WISHBONE bus cycle. As the name implies, it is used to transfer a single data operand. Figure A-7 shows a typical SINGLE READ cycle.

The WISHBONE specification shows all bus cycle timing diagrams as if the MASTER and SLAVE interfaces were connected in a point-to-point configuration. They also show all of the signals on the MASTER side of the interface. This provides a standard way of describing the interface without having to describe the whole system. For example, the Figure shows a signal called [ACK_I], which is an input to a MASTER interface. In this configuration it is directly connected to [ACK_O], which is driven by the SLAVE. If the timing diagram were shown from the perspective of the SLAVE, then the [ACK_O] signal would have been shown. The SINGLE READ cycle operates thusly:

1. In response to clock edge 0, the MASTER interface asserts [ADR_O()], [WE_O], [SEL_O], [STB_O] and [CYC_O].

2. The SLAVE decodes the bus cycle by monitoring its [STB_I] and address inputs, and presents valid data on its [DAT_O()] lines. Because the system is in a point-to-point configuration, the SLAVE [DAT_O()] signals are connected to the MASTER [DAT_I()] signals.

3. The SLAVE indicates that it has placed valid data on the data bus by asserting the MASTER's [ACK_I] acknowledge signal. Also note that the SLAVE may delay its response by inserting one or more wait states. In this case, the SLAVE does not assert the acknowledge line. The possibility of a wait state in the timing diagrams is indicated by '-WSS-'.

4. The MASTER monitors the state of its [ACK_I] line, and determines that the SLAVE has acknowledged the transfer at clock edge 1.

5. The MASTER latches [DAT_I()] and negates its [STB_O] signal in response to [ACK_I].

The SINGLE WRITE cycle operates in a similar manner, except that the MASTER asserts [WE_O] and places data on [DAT_O]. In this case the SLAVE asserts [ACK_O] when it has latched the data.
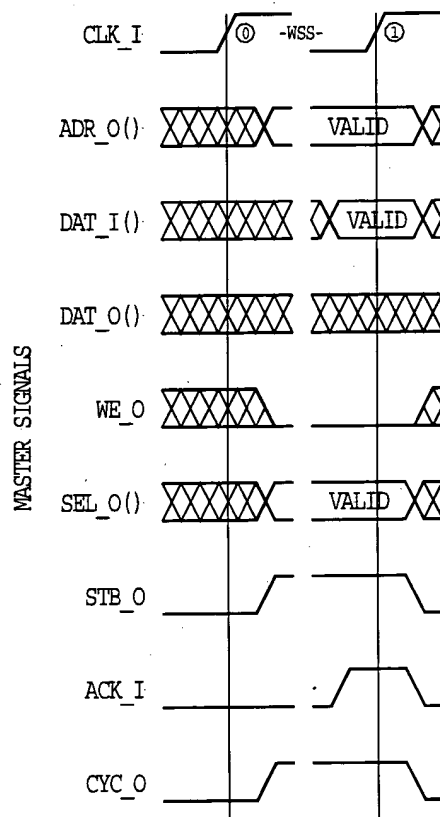
Figure A-7. SINGLE READ cycle.

A.4.2 BLOCK READ/WRITE Cycle

The BLOCK READ/WRITE cycles are very similar to the SINGLE READ/WRITE cycles. The BLOCK cycles can be thought of as two or more back-to-back SINGLE cycles strung together. In WISHBONE terminology the term *cycle* refers to the whole BLOCK cycle. The small, individual data transfers that make up the BLOCK cycle are called *phases*.

The starting and stopping point of the BLOCK cycles are identified by the assertion and negation of the MASTER [CYC_O] signal (respectively). The [CYC_O] signal is also used in shared bus and crossbar interconnections because it informs system logic that the MASTER wishes to use the bus. It also informs the interconnection when it is through with its bus cycle.

A.4.3 READ-MODIFY-WRITE (RMW) Cycle

The READ-MODIFY-WRITE cycle is used in multiprocessor and multitasking systems. This special cycle allows multiple software processes to share common resources by using semaphores. This is commonly done on interfaces for disk controllers, serial ports and memory. As the name implies, the READ-MODIFY-WRITE cycle reads and writes data to a memory location in a single bus cycle. It prevents the allocation of a common resource to two or more processes. The READ-MODIFY-WRITE cycle can also be thought of as an *indivisible cycle*.

The read portion of the cycle is called the *read phase*, and the write portion is called *the write phase*. When looking at the timing diagram of this bus cycle, it can be thought of as a two phase BLOCK cycle where the first phase is a read and the second phase is a write.

The READ-MODIFY-WRITE cycle is also known as an indivisible cycle because it is designed for multiprocessor systems. WISHBONE shared bus interconnections must be designed so that once an arbiter grants the bus to a first MASTER, it will not grant the bus to a second MASTER until the first MASTER gives up the bus. This allows a single MASTER (such as a microprocessor) to read some data, modify it and then write it back...all in a single, contiguous bus cycle. If the arbiter were allowed to change MASTERs in the middle of the cycle, then the two processors could incorrectly interpret the semaphore. The arbiter does this by monitoring the [CYC_O] cycle from each MASTER on the interconnection. The problem is averted because the [CYC_O] signal is always asserted for the duration of the RMW cycle.

To illustrate this point, consider a two processor system with a single disk controller. In this case each processor has a MASTER interface, and the disk controller has a SLAVE interface. Oftentimes, these systems require that only one processor access the disk at any given time[6]. To satisfy this requirement, a semaphore bit somewhere in memory is assigned to act as a 'traffic cop' between the two processors. If the bit is cleared, then the disk is available for use. If it's set, then the disk controller is busy.

---

[6] This is a common requirement to prevent one form of disk 'thrashing'. In this case, if both processors were allowed to access the disk during the same time interval, then one processor could request data from one sector of the disk while the other requested data from another sector. This could cause a situation where the disk head is constantly moved between the two locations, thereby degrading its performance or causing it to fail altogether.

Now consider a system where the two processors both need to use the disk. We'll call them processor #0 and processor #1. In order for processor #0 to acquire the disk it first reads and stores the state of the semaphore bit, and then sets the bit by writing back to memory. The reading and setting of the bit takes place inside of a single RMW cycle.

Once the processor is done with the semaphore operation, it checks the state of the bit it read during the first phase of the RMW cycle. If the bit is clear it goes ahead and uses the disk controller. If the other processor attempts to use the disk controller at this time, it reads a '1' from the semaphore, thereby preventing it from accessing the disk controller. When the first processor (#0) is done with the disk controller, it simply clears the semaphore bit by writing a '0' to it. This allows the other processor to gain access to the controller the next time it checks the semaphore.

Now consider the same situation, except where the semaphore is set and cleared using a SINGLE READ cycle followed by a SINGLE WRITE cycle. In this case it is possible for both processors to gain access to the disk controller at the same time...a situation that would crash the system. That's because the arbiter can grant the bus in the following order:

o   Processor #0 reads '0' from the semaphore bit.
o   Processor #1 reads '0' from the semaphore bit.
o   Processor #0 writes '1' to the semaphore bit.
o   Processor #1 writes '1' to the semaphore bit.

This leads to a system crash because both processors read a '0' from the semaphore bit, thereby causing both to access the disk controller.

It is important to note that a processor (or other device connected to the MASTER interface) must support the RMW cycle in order for this to be effective. This is generally done with special instructions that force a RMW bus cycle. Not all processors do this. A good example is the 680XX family of microprocessors. These use special compare-and-set (CAS) and test-and-set (TAS) instructions to generate RMW cycles, and to do the semaphore operations.

A.5 Endian

The WISHBONE specification regulates the ordering of data. This is because data can be presented in two different ways. In the first way, the most significant byte of an operand is placed at the higher (bigger) address. In the second way, the most significant byte of an operand can be placed at the lower (smaller) address. These are called BIG ENDIAN and LITTLE ENDIAN data operands, respectively. WISHBONE supports both types.

ENDIAN becomes an issue when the granularity of a WISHBONE port is smaller than the operand size. For example, a 32-bit port can have an 8-bit (BYTE wide) granularity. This results in a fairly ambiguous situation where the most significant byte of the 32-bit operand could be

placed at the higher or lower byte address of the port. However, ENDIAN is not an issue when the granularity and port size are the same.

The system integrator may wish to connect a BIG ENDIAN interface to a LITTLE ENDIAN inteface. In many cases the conversion is quite straightforward, and does not require any exotic conversion logic. Furthermore, the conversion does not create any speed degradation in the interface. In general, the ENDIAN conversion takes place by renaming the data and select I/O signals at a MASTER or SLAVE interface.

Figure A-8 shows a simple example where a 32-bit BIG ENDIAN MASTER output (CORE 'A') is connected to a 32-bit LITTLE ENDIAN SLAVE input (CORE 'B'). Both interfaces have 32-bit operand sizes and 8-bit granularities. As can be seen in the diagram, the ENDIAN conversion is accomplished by cross coupling the data and select signal arrays. This is quite simple since the conversion is accomplished at the interconnection level, or using a wrapper. This is especially simple in soft IP cores using VHDL or Verilog® hardware description languages, as it only requires the renaming of signals.

In some cases the address lines may also need to be modified between the two cores. For example, if 64-bit operands are transferred between two cores with 8-bit port sizes, then the address lines may need to be modified as well.
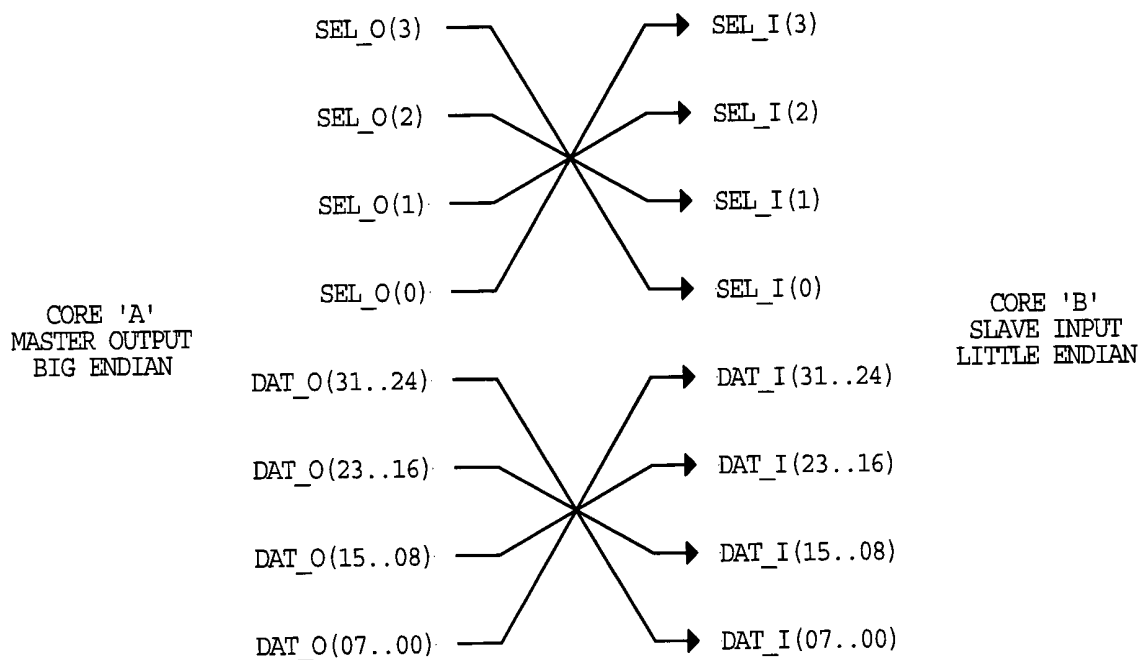
Figure A-8. BIG ENDIAN to LITTLE ENDIAN conversion example.

A.6 SLAVE I/O Port Examples

In this section we'll investigate several examples of WISHBONE interface for SLAVE I/O ports. Our purpose is to:

o   Show some simple examples of how the WISHBONE interface operates.
o   Demonstrate how simple interfaces work in conjunction with standard logic primitives on FPGA and ASIC devices. This also means that very little logic is needed to implement the WISHBONE interface.
o   Demonstrate the concept of *granularity*.
o   Provide some portable design examples.
o   Give examples of the WISHBONE DATASHEET.
o   Show VHDL implementation examples.

A.6.1 Simple 8-bit SLAVE Output Port

Figure A-9 shows a simple 8-bit WISHBONE SLAVE output port. The entire interface is implemented with a standard 8-bit 'D-type' flip-flop register (with synchronous reset) and a single AND gate. During write cycles, data is presented at the data input bus [DAT_I(7..0)], and is latched at the rising edge of [CLK_I] when [STB_I] and [WE_I] are both asserted.
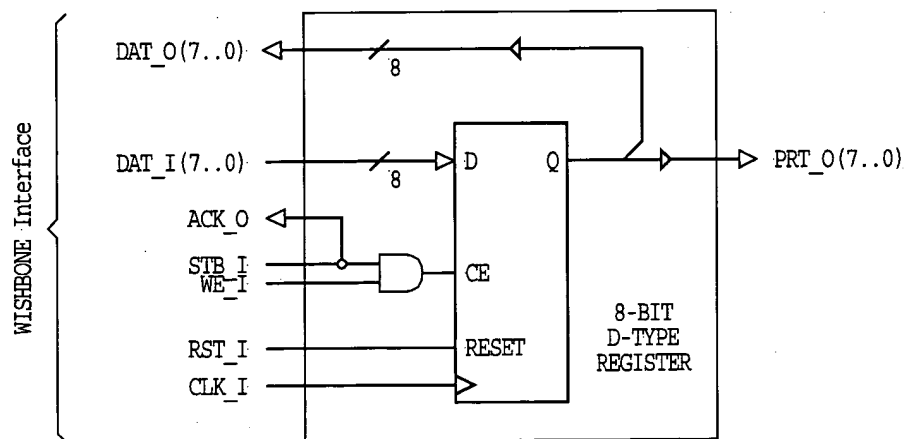
Figure A-9. Simple 8-bit WISHBONE SLAVE output port.

The state of the output port can be monitored by a MASTER by routing the output data lines back to [DAT_O(7..0)]. During read cycles the AND gate prevents erroneous data from being latched into the register.

This circuit is highly portable, as all FPGA and ASIC target devices support D-type flip-flops with clock enable and synchronous reset inputs.

The circuit also demonstrates how the WISHBONE interface requires little or no logic overhead. In this case, the WISHBONE interface does not require any extra logic gates whatsoever. This is because WISHBONE is designed to work in conjunction with standard, synchronous and combinatorial logic primitives that are available on most FPGA and ASIC devices.

The WISHBONE specification requires that the interface be documented. This is done in the form of the WISHBONE DATASHEET. The standard does not specify the form of the datasheet. For example, it can be part of a comment field in a VHDL or Verilog® source file or part of a technical reference manual for the IP core. Table A-1 shows one form of the WISHBONE DATASHEET for the 8-bit output port circuit.

The purpose of the WISHBONE DATASHEET is to promote design reuse. It forces the originator of the IP core to document how the interface operates. This makes it easier for another person to re-use the core.

Table A-1. WISHBONE DATASHEET for the 8-bit output port example.	
Description	Specification
General description:	8-bit SLAVE output port.
Supported cycles:	SLAVE, READ/WRITE SLAVE, BLOCK READ/WRITE SLAVE, RMW
Data port, size: Data port, granularity: Data port, maximum operand size: Data transfer ordering: Data transfer sequencing:	8-bit 8-bit 8-bit Big endian and/or little endian Undefined
Supported signal list and cross reference to equivalent WISHBONE signals:	Signal Name WISHBONE Equiv. ACK_O ACK_O CLK_I CLK_I DAT_I(7..0) DAT_I() DAT_O(7..0) DAT_O() RST_I RST_I STB_I STB_I WE_I WE_I

Figure A-10 shows a VHDL implementation of same circuit. The WBOPRT08 entity implements the all of the functions shown in the schematic diagram of Figure A-9.

```vhdl
library ieee;
use ieee.std_logic_1164.all;

entity WBOPRT08 is
port(
        -- WISHBONE SLAVE interface:

        ACK_O:      out     std_logic;
        CLK_I:      in      std_logic;
        DAT_I:      in      std_logic_vector( 7 downto 0 );
        DAT_O:      out     std_logic_vector( 7 downto 0 );
        RST_I:      in      std_logic;
        STB_I:      in      std_logic;
        WE_I:       in      std_logic;

        -- Output port (non-WISHBONE signals):

        PRT_O:      out     std_logic_vector( 7 downto 0 )

    );
end entity WBOPRT08;

architecture WBOPRT081 of WBOPRT08 is

    signal  Q: std_logic_vector( 7 downto 0 );

begin

    REG: process( CLK_I )
    begin

        if( rising_edge( CLK_I ) ) then

            if( RST_I = '1' ) then
                Q <= B"00000000";
            elsif( (STB_I and WE_I) = '1' ) then
                Q <= DAT_I( 7 downto 0 );
            else
                Q <= Q;
            end if;

        end if;

    end process REG;

    ACK_O <= STB_I;
    DAT_O <= Q;
    PRT_O <= Q;

end architecture WBOPRT081;
```

Figure A-10. VHDL implementation of the 8-bit output port interface.

A.6.2 Simple 16-bit SLAVE Output Port With 16-bit Granularity

Figure A-11 shows a simple 16-bit WISHBONE SLAVE output port. Table A-2 shows the WISHBONE DATASHEET for this design. It is identical to the 8-bit port shown earlier, except that the data bus is wider. Also, this port has 16-bit granularity. In the next section, it will be compared to a 16-bit port with 8-bit granularity.
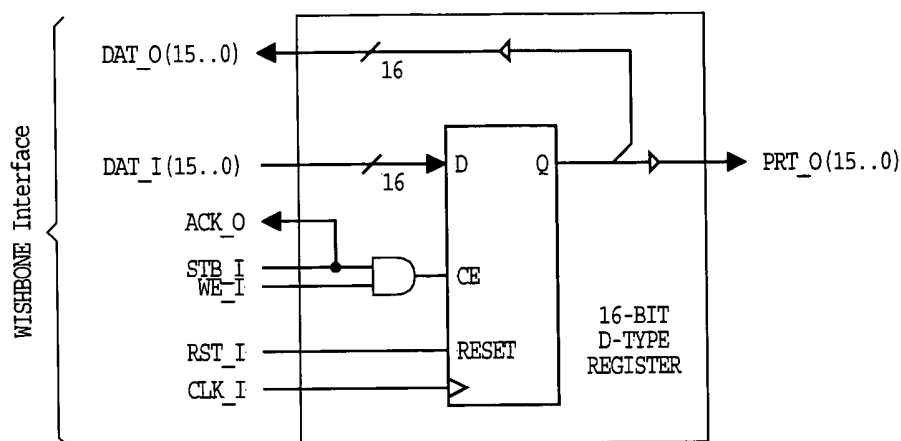
Figure A-11. Simple 16-bit WISHBONE SLAVE output port with 16-bit granularity

Table A-2. WISHBONE DATASHEET for the 16-bit output port with 16-bit granularity.	
Description	Specification
General description:	16-bit SLAVE output port.
Supported cycles:	SLAVE, READ/WRITE SLAVE, BLOCK READ/WRITE SLAVE, RMW
Data port, size: Data port, granularity: Data port, maximum operend size: Data transfer ordering: Data transfer sequencing:	16-bit 16-bit 16-bit Big endian and/or little endian Undefined
Supported signal list and cross reference to equivalent WISHBONE signals:	Signal Name WISHBONE Equiv. ACK_O ACK_O CLK_I CLK_I DAT_I(15..0) DAT_I() DAT_O(15..0) DAT_O() RST_I RST_I STB_I STB_I WE_I WE_I

A.6.3 Simple 16-bit SLAVE Output Port With 8-bit Granularity

Figure A-12 shows a simple 16-bit WISHBONE SLAVE output port. This port has 8-bit granularity, which means that data can be transferred 8 or 16-bits at a time.
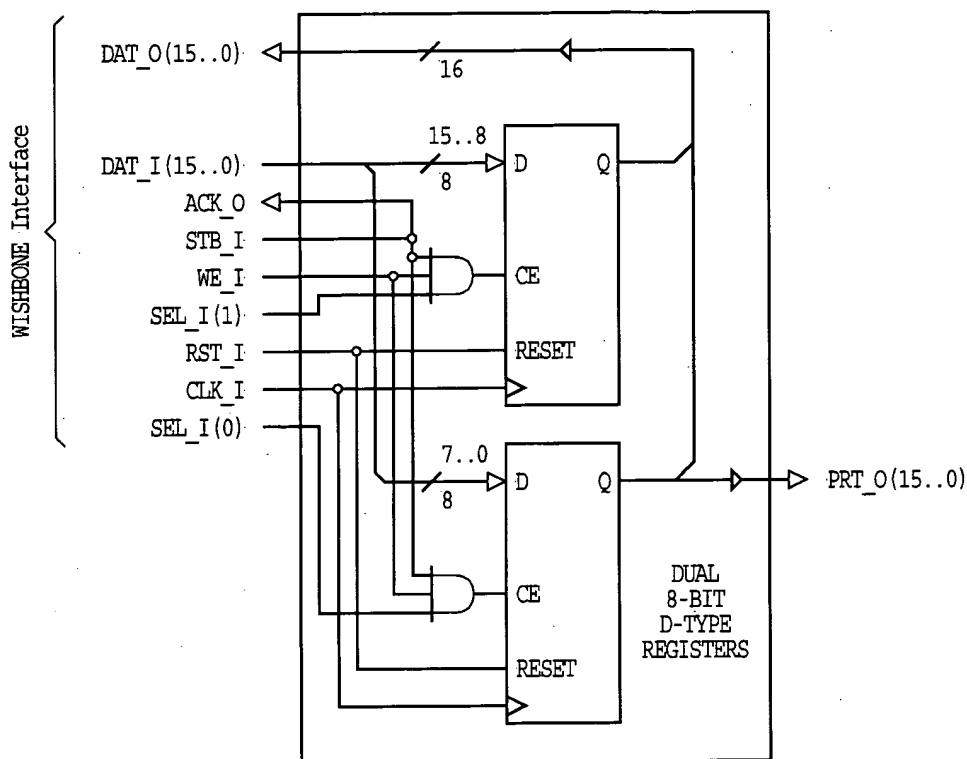
Figure A-12. Simple 16-bit WISHBONE SLAVE output port with 8-bit granularity.

This circuit differs from the aforementioned 16-bit port because it has 8-bit granularity. This means that the 16-bit register can be accessed with either 8 or 16-bit bus cycles. This is accomplished by selecting the high or low byte of data with the select lines [SEL_I(1..0)]. When [SEL_I(0)] is asserted, the low byte is accessed. When [SEL_I(1)] is asserted, the high byte is accessed. When both are asserted, the entire 16-bit word is accessed.

The circuit also demonstrates the proper use of the [STB_I] and [SEL_I()] lines for slave devices. The [STB_I] signal operates much like a chip select signal, where the interface is selected when [STB_I] is asserted. The [SEL_I()] lines are only used to determine where data is placed by the MASTER or SLAVE during read and write cycles.

In general, the [SEL_I()] signals should never be used by the SLAVE to determine when the IP core is being accessed by a master. They should only be used to determine where data is placed on the data input and output buses. Stated another way, the MASTER will assert the select signals [SEL_O()] during every bus cycle, but a particular slave is only accessed when it monitors

that its [STB_I] input is asserted. Stated another way, the [STB_I] signal is generated by address decode logic within the WISHBONE interconnect, but the [SEL_I()] signals may be broadcasted to all SLAVE devices.

Table A-3 shows the WISHBONE DATASHEET for this IP core. This is very similar to the 16-bit data port with 16-bit granularity, except that the granularity has been changed to 8-bits.

It should also be noted that the datasheet specifies that the circuit will work with READ/WRITE, BLOCK READ/WRITE and RMW cycles. This means that the circuit will operate normally when presented with these cycles. It is left as an exercise for the user to verify that the circuit will indeed work with all three of these cycles.

Table A-3. WISHBONE DATASHEET for the 16-bit output port with 8-bit granularity.	
Description	Specification
General description:	16-bit SLAVE output port with 8-bit granularity.
Supported cycles:	SLAVE, READ/WRITE SLAVE, BLOCK READ/WRITE SLAVE, RMW
Data port, size: Data port, granularity: Data port, maximum operand size: Data transfer ordering: Data transfer sequencing:	16-bit 8-bit 16-bit Big endian and/or little endian Undefined
Supported signal list and cross reference to equivalent WISHBONE signals:	Signal Name WISHBONE Equiv. ACK_O ACK_O CLK_I CLK_I DAT_I(15..0) DAT_I() DAT_O(15..0) DAT_O() RST_I RST_I STB_I STB_I WE_I WE_I

Figure A-13 shows a VHDL implementation of same circuit. The WBOPRT16 entity implements the all of the functions shown in the schematic diagram of Figure A-12.

```vhdl
entity WBOPRT16 is
port(
        -- WISHBONE SLAVE interface:
        ACK_O:      out    std_logic;
        CLK_I:      in     std_logic;
        DAT_I:      in     std_logic_vector( 15 downto 0 );
        DAT_O:      out    std_logic_vector( 15 downto 0 );
        RST_I:      in     std_logic;
        SEL_I:      in     std_logic_vector(  1 downto 0 );
        STB_I:      in     std_logic;
        WE_I:       in     std_logic;

        -- Output port (non-WISHBONE signals):
        PRT_O:      out    std_logic_vector( 15 downto 0 )

    );
end entity WBOPRT16;

architecture WBOPRT161 of WBOPRT16 is
    signal  QH: std_logic_vector( 7 downto 0 );
    signal  QL: std_logic_vector( 7 downto 0 );
begin

    REG: process( CLK_I )
    begin
        if( rising_edge( CLK_I ) ) then
            if( RST_I = '1' ) then
                QH <= B"00000000";
            elsif( (STB_I and WE_I and SEL_I(1)) = '1' ) then
                QH <= DAT_I( 15 downto 8 );
            else
                QH <= QH;
            end if;
        end if;

        if( rising_edge( CLK_I ) ) then
            if( RST_I = '1' ) then
                QL <= B"00000000";
            elsif( (STB_I and WE_I and SEL_I(0)) = '1' ) then
                QL <= DAT_I( 7 downto 0 );
            else
                QL <= QL;
            end if;
        end if;

    end process REG;

    ACK_O <= STB_I;
    DAT_O( 15 downto 8 )  <= QH;
    DAT_O(  7 downto 0 )  <= QL;
    PRT_O( 15 downto 8 )  <= QH;
    PRT_O(  7 downto 0 )  <= QL;

end architecture WBOPRT161;
```

Figure A-13. VHDL implementation of the 16-bit output port with 8-bit granularity.

A.7 WISHBONE Memory Interfacing

In this section we'll examine WISHBONE memory interfacing and present some examples. The purpose of this section is to:

- Introduce the FASM synchronous RAM and ROM models.
- Show a simple example of how the WISHBONE interface operates.
- Demonstrate how simple interfaces work in conjunction with standard logic primitives on FPGA and ASIC devices. This also means that very little logic (if any) is needed to implement the WISHBONE interface.
- Present a WISHBONE DATASHEET example for a memory element.
- Describe portability issues with regard to FPGA and ASIC memory elements.

A.7.1 FASM Synchronous RAM and ROM Model

The WISHBONE interface can be connected to any type of RAM or ROM memory. However, some types will be faster and more efficient than others. If the memory interface closely resembles the WISHBONE interface, then everything will run very fast. If the memory is significantly different than WISHBONE, then everything will slow down. This is such a fundamental and important issue that both the WISHBONE interface and its bus cycles were designed around the most common memory interface that could be found.

This was very problematic in the original WISHBONE concept. That's because there are very few portable RAM and ROM types used across all both FPGA and ASIC devices. In fact, the most common memory type that could be found are what we call 'FASM', or the FPGA and ASIC Subset Model[7].

The FASM synchronous RAM model conforms to the connection and timing diagram shown in Figure A-14. The WISHBONE bus cycles all are designed to interface directly to this type of RAM. During write cycles, FASM Synchronous RAM stores input data at the indicated address whenever: (a) the write enable (WE) input is asserted, and (b) there is a rising clock edge.

During read cycles, FASM Synchronous RAM works like an asynchronous ROM. Data is fetched from the address indicated by the ADR() inputs, and is presented at the data output (DOUT). The clock input is ignored. However, during write cycles, the output data is updated immediately during a write cycle.

A good exercise for the user is to compare the FASM Synchronous RAM cycles to the WISHBONE SINGLE READ/WRITE, BLOCK READ/WRITE and READ-MODIFY-WRITE cycles. You will find that these three bus cycles operate in an identical fashion to the FASM Synchronous RAM model. They are so close, in fact, that FASM RAMs can be interfaced to WISHBONE with as little as one NAND gate.

---

[7] The original FASM model actually encompasses many type of devices, but in this tutorial we'll focus only on the FASM synchronous RAM and ROM models.

While most FPGA and ASIC devices will provide RAM that follows the FASM guidelines, you will probably find that most devices also support other types of memories as well. For example, in some brands of FPGA you will find block memories that use a different interface. Some of these will still interface very smoothly to WISHBONE, while others will introduce a wait-state. In all cases that we found, all FPGAs and most ASICs did support at least one style of FASM memory.
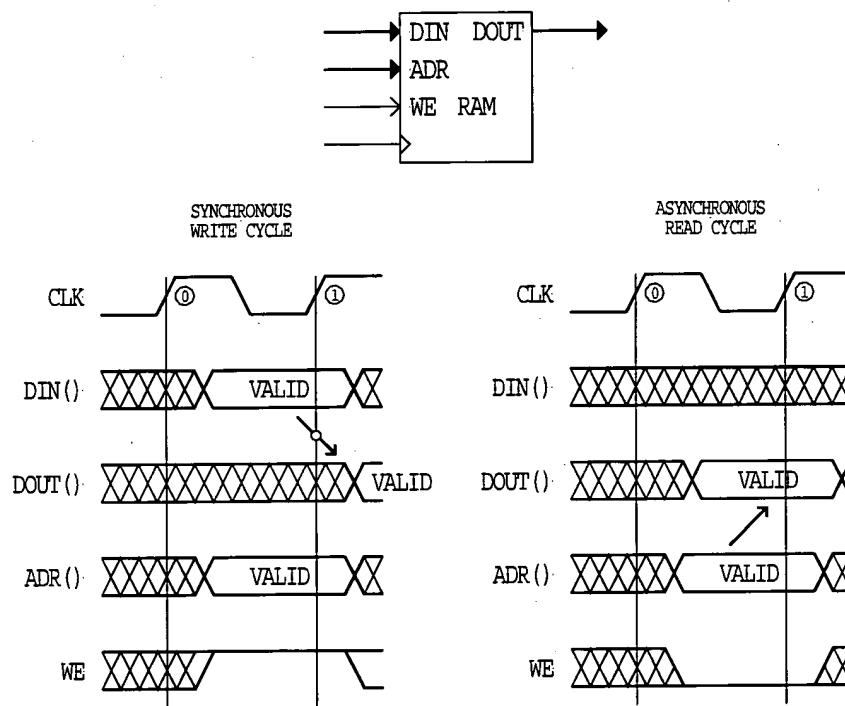
Figure A-14. Generic FASM synchronous RAM connection and timing diagram.

The FASM ROM closely resembles the FASM RAM during its read cycle. FASM ROM conforms to the connection and timing diagram shown in Figure A-15.
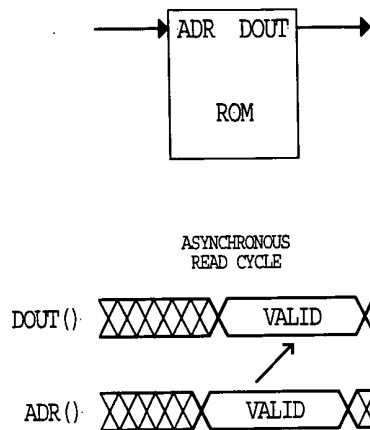
Figure A-15. FASM asynchronous ROM connection and timing diagram.

A.7.2 Simple 16 x 8-bit SLAVE Memory Interface

Figure A-16 shows a simple 8-bit WISHBONE memory. The 16 x 8-bit memory is formed from two 16 x 4-bit FASM compatible synchronous memories. Besides the memory elements, the entire interface is implemented with a single AND gate. During write cycles, data is presented at the data input bus [DAT_I(7..0)], and is latched at the rising edge of [CLK_I] when [STB_I] and [WE_I] are both asserted. During read cycles, the memory output data (DO) is made available at the data output port [DAT_O(7..0)].
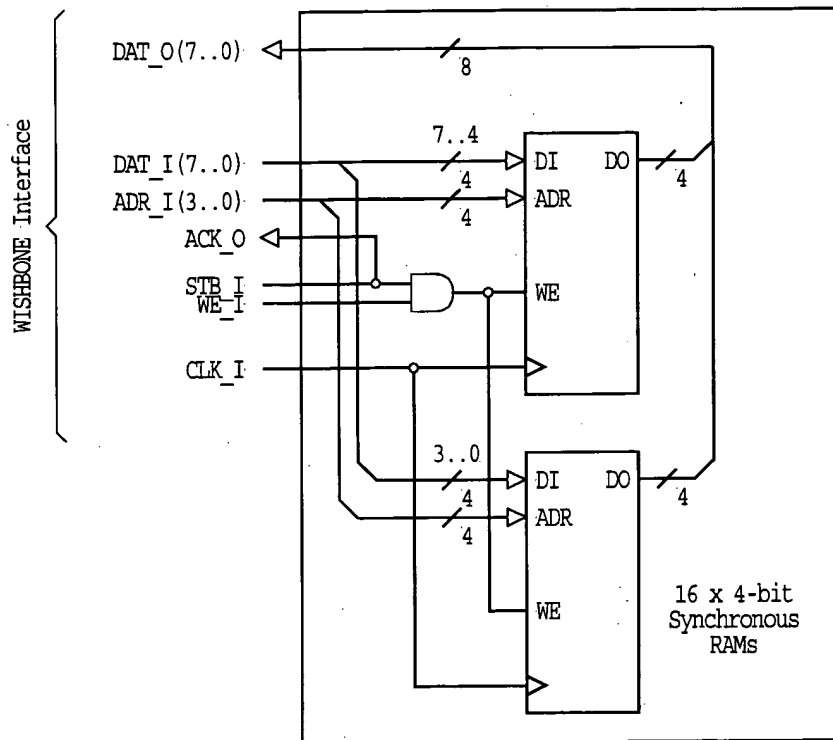
Figure A-16. Simple 16 x 8-bit SLAVE memory.

The memory circuit does not have a reset input. That's because most RAM memories do not have a reset capability.

The circuit also demonstrates how the WISHBONE interface requires little or no logic overhead. In this case, the WISHBONE interface requires a single AND gate. This is because WISHBONE is designed to work in conjunction with standard, synchronous and combinatorial logic primitives that are available on most FPGA and ASIC devices.

The WISHBONE specification requires that the interface be documented. This is done in the form of the WISHBONE DATASHEET. The standard does not specify the form of the datasheet. For example, it can be part of a comment field in a VHDL or Verilog® source file or part of a technical reference manual for the IP core. Table A-4 shows one form of the WISHBONE DATASHEET for the 16 x 8-bit memory IP core.

The purpose of the WISHBONE DATASHEET is to promote design reuse. It forces the originator of the IP core to document how the interface operates. This makes it easier for another person to re-use the core.

Table A-4. WISHBONE DATASHEET for the 16 x 8-bit SLAVE memory.	
Description	Specification
General description:	16 x 8-bit memory IP core.
Supported cycles:	SLAVE, READ/WRITE SLAVE, BLOCK READ/WRITE SLAVE, RMW
Data port, size: Data port, granularity: Data port, maximum operand size: Data transfer ordering: Data transfer sequencing:	8-bit 8-bit 8-bit Big endian and/or little endian Undefined
Clock frequency constraints:	NONE (determined by memory primitive)
Supported signal list and cross reference to equivalent WISHBONE signals:	Signal Name WISHBONE Equiv. ACK_O ACK_O ADR_I(3..0) ADR_I() CLK_I CLK_I DAT_I(7..0) DAT_I() DAT_O(7..0) DAT_O() STB_I STB_I WE_I WE_I
Special requirements:	Circuit assumes the use of synchronous RAM primitives with asynchronous read capability.

A.7.3 Memory Primitives and the [ACK_O] Signal

Memory primitives, specific to the FPGA or ASIC target device, are usually used for the RAM storage elements. That's because most high-level languages (such as VHDL and Verilog®) don't synthesize these very efficiently. For this reason, the user should verify that the memory primitives are available for the target device.

The memory circuits shown above are highly portable, but do assume that FASM compatible memories are available. During *write* cycles, most synchronous RAM primitives latch the input data when at the rising clock edge when the write enable input is asserted. However, during *read* cycles the RAM primitives may behave in different ways.

There are two types of RAM primitives that are generally found on FPGA and ASIC devices: (a) those that synchronously present data at the output after the rising edge of the clock input, and (b) those that asynchronously present data at the output after the address is presented to the RAM element.

The circuit assumes that the RAM primitive is the FASM, asynchronous read type. That's because during read cycles the WISHBONE interface assumes that output data is valid at the rising

[CLK_I] edge following the assertion of the [ACK_O] output. Since the circuit ties the [STB_I] signal back to the [ACK_O] signal, the asynchronous read RAM is needed on the circuit shown here.

For this reason, if *synchronous* read type RAM primitives are used, then the circuit must be modified to insert a single wait-state during all read cycles. This is quite simple to do, and only requires an additional flip-flop and gate in the [ACK_O] circuit.

Furthermore, it can be seen that the circuit operates faster if the asynchronous read type RAM primitives are used. That's because the [ACK_O] signal can be asserted immediately after the assertion of [STB_I]. If the synchronous read types are used, then a single-clock wait-state must be added.

A.8 Point-to-point Interconnection Example

Now that we've reviewed some of the WISHBONE basics, it's time to try them out with a simple example. In this section we'll create a complete WISHBONE system with a point-to-point interconnection. The system includes a 32-bit MASTER interface to a DMA[8] unit, and a 32-bit SLAVE interface to a memory. In this example the DMA transfers data to and from the memory using block transfer cycles.

The purpose of this system is to create a portable benchmarking device. Although the system is very simple, it does allow the user to determine the typical maximum speeds and minimum sizes on any given FPGA or ASIC target device[9].

Source code for this example can be found in the <u>WISHBONE Public Domain Library for VHDL</u> (under 'EXAMPLE1' in the EXAMPLES folder). The library also has detailed documentation for the library modules, including detailed circuit descriptions and timing diagrams for the INTERCON, SYSCON, DMA and memory interfaces. The reader is encouraged to review and experiment with all of these files.

Figure A-17 shows the system. The WISHBONE interconnection system (INTERCON) can be found in file ICN0001a. That system connects a simple DMA MASTER (DMA0001a) to an 8 x 32-bit register based memory SLAVE (MEM0002a). The reset and clock signals are generated by system controller SYSCON (SYC0001a).

---

[8] DMA: Direct Memory Access.
[9] Benchmarking can be a difficult thing to do. On this system the philosophy was to create a 'real-world' SoC that estimates 'typical maximum' speeds and 'typical minimum' size. It's akin to the 'flight envelope' of an airplane. A flight envelope is a graph that shows the altitude vs. the speed of the aircraft. It's one 'benchmark' for the airplane. While the graph may show that the airplane is capable of flying at MACH 2.3 at an altitude of 28,000 meters, it may never actually fly in that situation during its lifetime. The graph is simply a tool for quickly understanding the engineering limits of the design. The same is true for the WISHBONE benchmarks given in this tutorial. However, having said this it is important to remember that the benchmarks are real systems, and do include all of the logic and routing resources needed to implement the design.
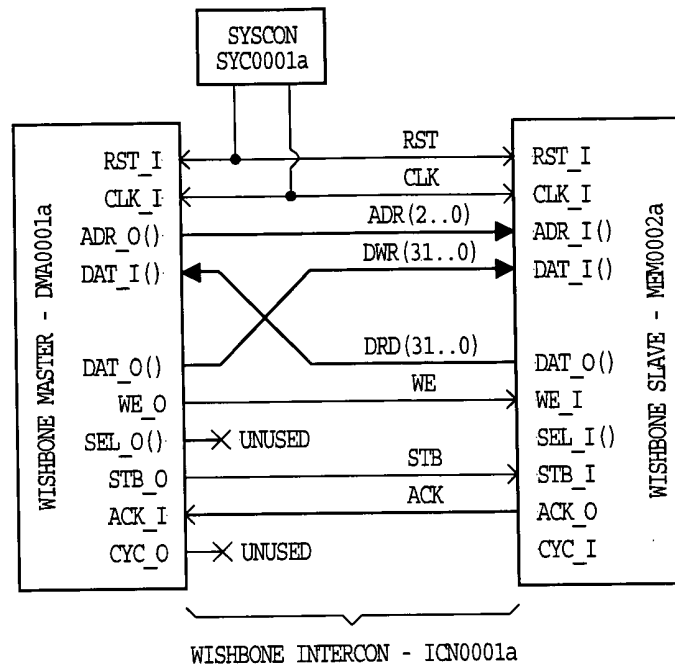
Figure A-17. Point-to-point interconnection example.

This system was synthesized and routed on two styles of Xilinx[10] FPGA: the Spartan 2 and the Virtex 2. For benchmarking purposes the memories were altered so that they used Xilinx distributed RAMs instead of the register RAMs in MEM0002a. A memory interface for the Xilinx RAMs can be found in MEM0001a, which is substituted for MEM0002a.

It should be noted that the Xilinx distributed RAMs are quite efficient on the WISHBONE interface. As can be seen in the source code, only a single 'AND' gate was needed to interface the RAM to WISHBONE.

The system for the Xilinx Spartan 2 was synthesized and operated on a Silicore evaluation board. This was a 'reality check' that verified that things actually routed and worked as expected. Some of the common signals were brought out to test points on the evaluation board. These were monitored with an HP54620a logic analyzer to verify the operation. Figure A-18 shows an example trace from the logic analyzer. Address lines, data write lines and several control signals were viewed. That Figure shows a write cycle with eight phases followed by a read cycle with eight phases. The data lines always show 0x67 because that's the data transferred by the DMA in this example.

---

[10] Xilinx is a registered trademark of Xilinx, Inc.

Sampled @ 4.0S   ← 0.00s 500S/   Sngl PatSTOP

```
0 CLK
1 ESTB
2 EACK
3 EWE
4 EADR0
5 EADR1
6 EADR2
15 EDWR7
14 EDWR6
13 EDWR5
12 EDWR4
11 EDWR3
10 EDWR2
9 EDWR1
8 EDWR0
```

Trig= 15 XXXX XXXX XXXX XXXX 0 Ext X
Print   Hardcopy   I/O   Service
Screen   Menu   Menu   Menu
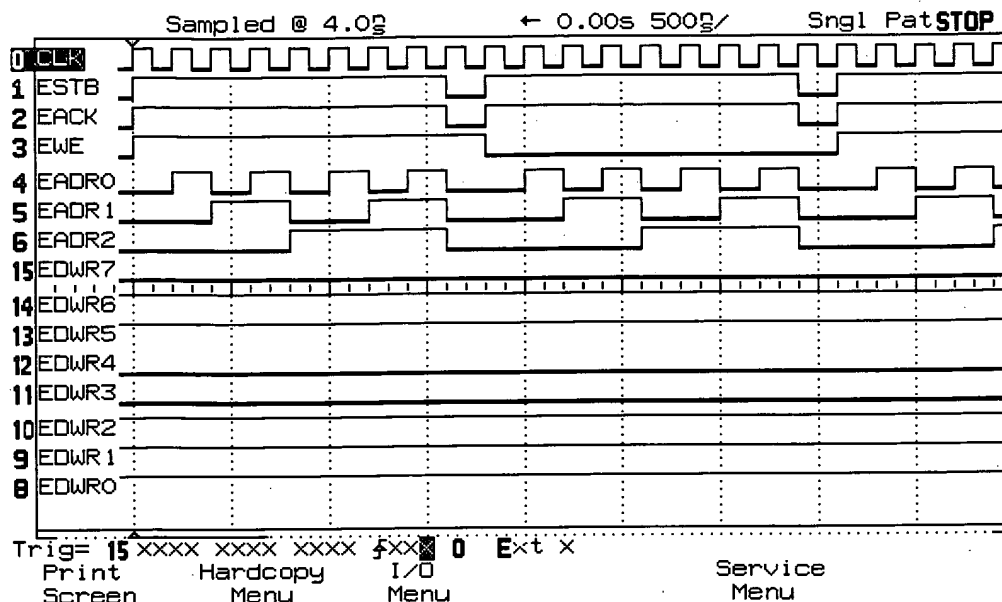
Figure A-18. Logic analyzer trace on the Spartan 2 evaluation board[11].

Table A-5 shows the speed of the system after synthesis and routing. The Spartan 2 bench-marked at about 428 Mbyte/sec, and was tested in hardware (HW TEST). The Virtex 2 part was synthesized and routed, but was only tested under software (SW TEST).

Table A-5. 32-bit Point-to-point Interconnection Benchmark Results							
MFG & Type	Part Number	HW TEST	SW TEST	Size	Timing Constraint (MIN)	Actual Speed (MAX)	Data Transfer Rate (MAX)
Xilinx Spartan 2 (FPGA)	XC2S50-5-PQ208C	√		53 SLICE	100 MHz	107 MHz	428 Mbyte/sec
Xilinx Virtex 2 (FPGA)	XC2V40-4-FG256C		√	53 SLICE	200 MHz	203 MHz	812 Mbyte/sec

Notes:
VHDL synthesis tool: Altium Accolade PeakFPGA 5.30a
Router: Xilinx Alliance 3.3.06I_V2_SE2
Hardware evaluation board: Silicore 170101-00 Rev 1.0
Listed size was reported by the router.
Spartan 2 test used '-5' speed grade part (slower than the faster '-6' part).

[11] The logic analyzer samples at 500 Mhz, so the SoC was slowed down to make the traces look better. This trace was taken with a SoC clock speed of 5 MHz. Slowing the clock down is also a good way to verify that the speed of the WISHBONE interface can be 'throttled' up and down.

A.9 Shared Bus Example

Now that we've built a WISHBONE point-to-point interconnection, it's time to look at a more complex SoC design. In this example, we'll create a 32-bit shared bus system with four MAS-TERs and four SLAVEs. Furthermore, we'll re-use the same DMA, memory and SYSCON interfaces that we used in the point-to-point interconnection example above. This will demonstrate how WISHBONE interfaces can be adapted to many different system topologies.

This example will require the introduction of some new concepts. As the system integrator, we'll need to make some decisions about how we want our system to work. After that, we'll need to create the various parts of the design in order to finish the job. Some of the decisions and tasks include:

- Choosing between multiplexed and non-multiplexed bus topology.
- Choosing between three-state and multiplexor based interconnection logic.
- Creating the interconnection topology.
- Creating an address map (using variable address decoding).
- Creating a four level round-robin arbiter.
- Creating and benchmarking the system.

A.9.1 Choosing Between Multiplexed and Non-multiplexed Bus Topology

The first step in designing a shared bus is to determine how we'll move our data around the system. In this section we'll explore multiplexed and non-multiplexed buses, and explore some of the trade-offs between them.

The big advantage of multiplexed buses is that they reduce the number of interconnections by routing different types of data over the same set of signal lines. The most common form of multiplexed bus is where address and data lines share a common set of signals. A multiplexed bus is shown in Figure A-19. For example, a 32-bit address bus and 32-bit data bus can be combined to form a 32-bit common address/data bus. This reduces the number of routed signals from 64 to 32.

The major disadvantage of the multiplexed bus is that it takes twice as long to move the information. In this case a non-multiplexed, synchronous bus can generally move address and data information in as little as one clock cycle. Multiplexed address and data buses require at least two clock cycles to move the same information.

Since we're creating a benchmarking system that is optimized for speed, we'll use the non-multiplexed scheme for this example. This will allow us to move one data operand during every clock cycle.

It should be noted that multiplexed buses have long been used in the electronics industry. In semiconductor chips the technique is used to reduced the number of pins on a chip. In the mi-

crocomputer bus industry the technique is often used to reduce the number of backplane connector pins.
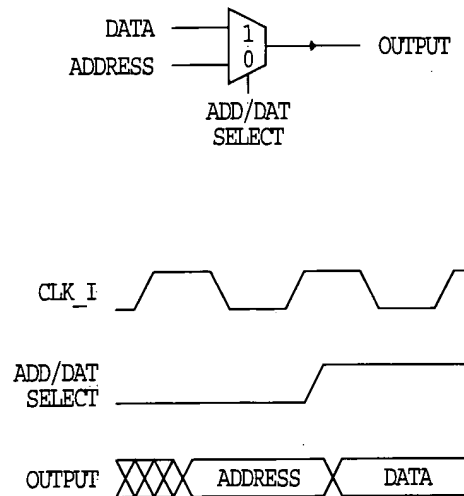
Figure A-19. Circuit and timing diagram for a multiplexed address/data bus.

A.9.2 Choosing Between Three-State and Multiplexor Interconnection Logic

WISHBONE interconnections can use three-state[12] or multiplexor logic to move data around a SoC. The choice depends on what makes sense in the application, and what's available on the integrated circuit.

Three-state I/O buffers have long been used in the semiconductor and microcomputer bus industries. These reduce the number of signal pins on an interface. In microcomputer buses with master-slave architectures, the master that 'owns' the bus turns its buffers 'on', while the other master(s) turn their buffers 'off'[13]. This prevents more than one bus master from driving any signal line at any given time. A similar situation also occurs at the slave end. There, a slave that participates in a bus cycle enables its output buffers during read cycles.

In WISHBONE, all IP core interfaces have 'in' and 'out' signals on the address, data and other internal buses. This allows the interface to be adapted to point-to-point, multiplexed and three-state I/O interconnections. Figure A-20 shows how the 'in' and 'out' signals can be connected to a three-state I/O bus[14].

---

[12] Three-state buffers are sometimes called Tri-State® buffers. Tri-State® is a registered trademark of National Semiconductor Corporation.

[13] Here, 'on' and 'off' refer to the three-state and non three-state conditions, respectively.

[14] Also note that the resistor/current source shown in the figure can also be a pull-down resistor or a bus terminator, or eliminated altogether.

Vcc

RESISTOR
-OR-
CURRENT
SOURCE

THREE-STATE CONTROL

DAT_O(0)

D   Q
CE

THREE-STATE
BUFFER

DAT_IO(0)

THREE-STATE
INTERCONNECT

CLK_I

DAT_I(0)

Q   D
CE

SIGNAL
NAME
TRANSLATION
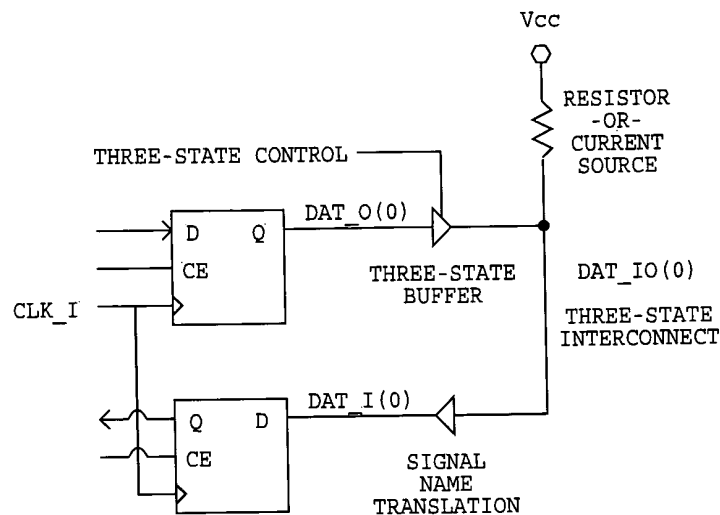
Figure A-20. Connection of a data bus bit to a three-state interconnection.

A simple SoC interconnection that uses three-state I/O buffers is shown in the block diagram of Figure A-21(a). There, the data buses on two master and two slave modules are interconnected with three-state logic. However, this approach has two major drawbacks. First, it is inherently slower than direct interconnections. That's because there are always minimum timing parameters that must be met to turn buffers on-and-off. Second, many IC devices do not have any internal three-state routing resources available to them, or they are very restrictive in terms of location or quantity of these interconnects.

As shown in Figure A-21(b), the SoC bus can be adapted to use multiplexor logic to achieve the same goal. The main advantage of this approach is that it does not use the three-state routing resources which are not available on many FPGA and ASIC devices.

The main disadvantage of the multiplexor logic interconnection is that it requires a larger number of routed interconnects and logic gates (which are not required with the three-state bus approach).

However, there is also a growing body of evidence that suggests that this type of interconnection is easier to route in both FPGA and ASIC devices. Although this is very difficult to quantify, the author has found that the multiplexor logic interconnection is quite easily handled by standard FPGA and ASIC routers. This is because:

- Three-state interconnections force router software to organize the SoC around the fixed three-state bus locations. In many cases, this constraint results in poorly optimized and/or slow circuits.

o Very often, 'bit locations' within a design are grouped together. In many applications, the multiplexor logic interconnection is easier to handle for place & route tools.

o Pre-defined, external I/O pin locations are easier to achieve with multiplexor logic interconnections. This is especially true with FPGA devices.

For the shared bus example we will use multiplexor logic for the interconnection. That's because multiplexor logic interconnections are more portable than three-state logic designs. The shared bus design in this example is intended to be used on many brands of FPGA and ASIC devices.
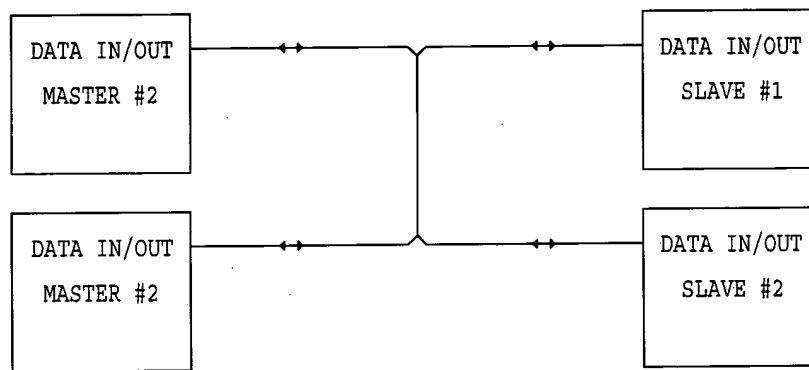
(A) THREE-STATE BUS INTERCONNECTION
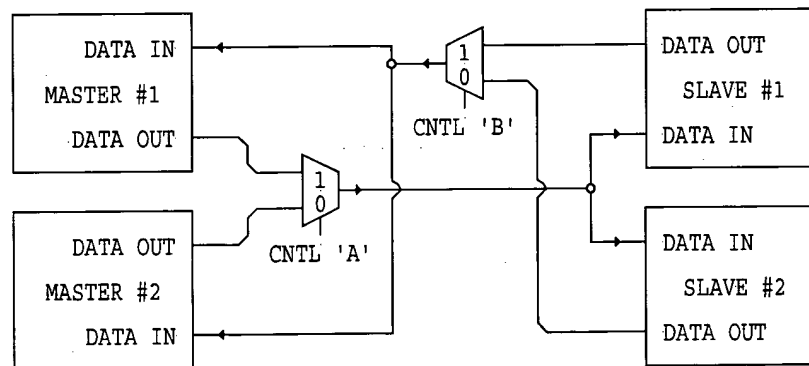
(B) MULTIPLEXOR LOGIC INTERCONNECTION

Figure A-21. Three-state bus interconnection vs. multiplexor logic interconnection.

A.9.3 Creating the Interconnection Topology

In the previous two sections it was decided to use multiplexor interconnections with non-multiplexed address and data buses. In this section we'll refine those concepts into a broad interconnection topology for our system. However, we'll save the details for later. For now, we're just interested in looking at some of the general issues.

In WISHBONE nomenclature, the interconnection is also called the INTERCON. The INTERCON is an IP Core that connects all of the MASTER, SLAVE and SYSCON cores together.

Figure A-22 shows the generic topology of an INTERCON that supports multiplexor interconnections with multiplexed address and data buses. By 'generic', we mean that there are 'N' MASTERs and SLAVEs shown in the diagram. The actual number of MASTER and SLAVE interfaces can be adjusted up or down, depending upon what's needed in the system. In the shared bus example we'll use four MASTERs and four SLAVEs. However, for now we'll think in more general terms.

An interface called the SYSCON provides the system with a stable clock [CLK_O] and reset signal [RST_O]. For now, we'll assume that the clock comes from off-chip, and that the reset signal is synchronized from some global system reset.

After the initial system reset, one or more MASTERs request the interconnection, which we'll call a 'bus' for now. MASTERs do this by asserting their [CYC_O] signal. An arbiter, which we'll discuss shortly, determines which MASTER can use the bus. One clock edge after the assertion of a [CYC_O] signal the arbiter grants the bus to one of the MASTERs that requested it. It grants the bus by asserting grant lines GNT0 – GNTN and GNT(N..0). This informs both the INTERCON as to which MASTER can own the bus.

Once the bus is arbitrated, the output signals from the selected MASTER are routed, via multiplexors, onto the shared buses. For example, if MASTER #0 obtains the bus, then the address lines [ADR_O()] from MASTER #0 are routed to shared bus [ADR()]. The same thing happens to the data out [DAT_O()], select out [SEL_O()], write enable [WE_O] and strobe [STB_O] signals. The shared bus output signals are routed to the inputs on the SLAVE interfaces.

The arbiter grant lines are also used to enable the terminating signals [ACK_I], [RTY_I] and [ERR_I]. These are enabled at the MASTER that acquired the bus. For example, if MASTER #0 is granted the bus by the arbiter, then the [ACK_I], [RTY_I] and [ERR_I] are enabled at MASTER #0. Other MASTERs, who may also be requesting the bus, never receive a terminating signal, and therefore will wait until they are granted the bus.

During this interval the common address bus [ADR()] is driven with the address lines from the MASTER. The address lines are decoded by the address comparator, which splits the address space into 'N' sections. The decoded output from the comparator is used to select the slave by way of its strobe input [STB_I]. A SLAVE may only respond to a bus cycle when its [STB_I] is asserted. This is also a wonderful illustration of the partial address decoding technique used by WISHBONE, which we'll discuss in depth below.

For example, consider a system with an addressing range of sixteen bits. If the addressing range were evenly split between all of the SLAVEs, then each SLAVE would be allocated 16 Kbytes of address space. This is shown in the address map of Figure A-23. In this case, the address comparator would decode bits [ADR(15..14)]. In actual practice the system integrator can alter the address map at his or her discretion.

Once a SLAVE is selected, it participates in the current bus cycle generated by the MASTER. In response to the cycle, the SLAVE must assert either its [ACK_O], [RTY_O] or [ERR_O] output. These signals are collected with an 'or' gate, and routed to the current MASTER.

Also note that during read cycles, the SLAVE places data on its [DAT_O()] bus. These are routed from the participating SLAVE to the current MASTER by way of a multiplexor. In this case, the multiplexor source is selected by some address lines which have been appropriately selected to switch the multiplexor.

Once the MASTER owning the bus has received an asserted terminating signal, it terminates the bus cycle by negating its strobe output [STB_O]. If the MASTER is in the middle of a block transfer cycle, it will generate another phase of the block transfer. If it's performing a SINGLE cycle, or if its at the end of a BLOCK cycle, the MASTER terminates the cycle by negating its [CYC_O] signal. This informs the MASTER that it's done with the bus, and that it can re-arbitrate it.
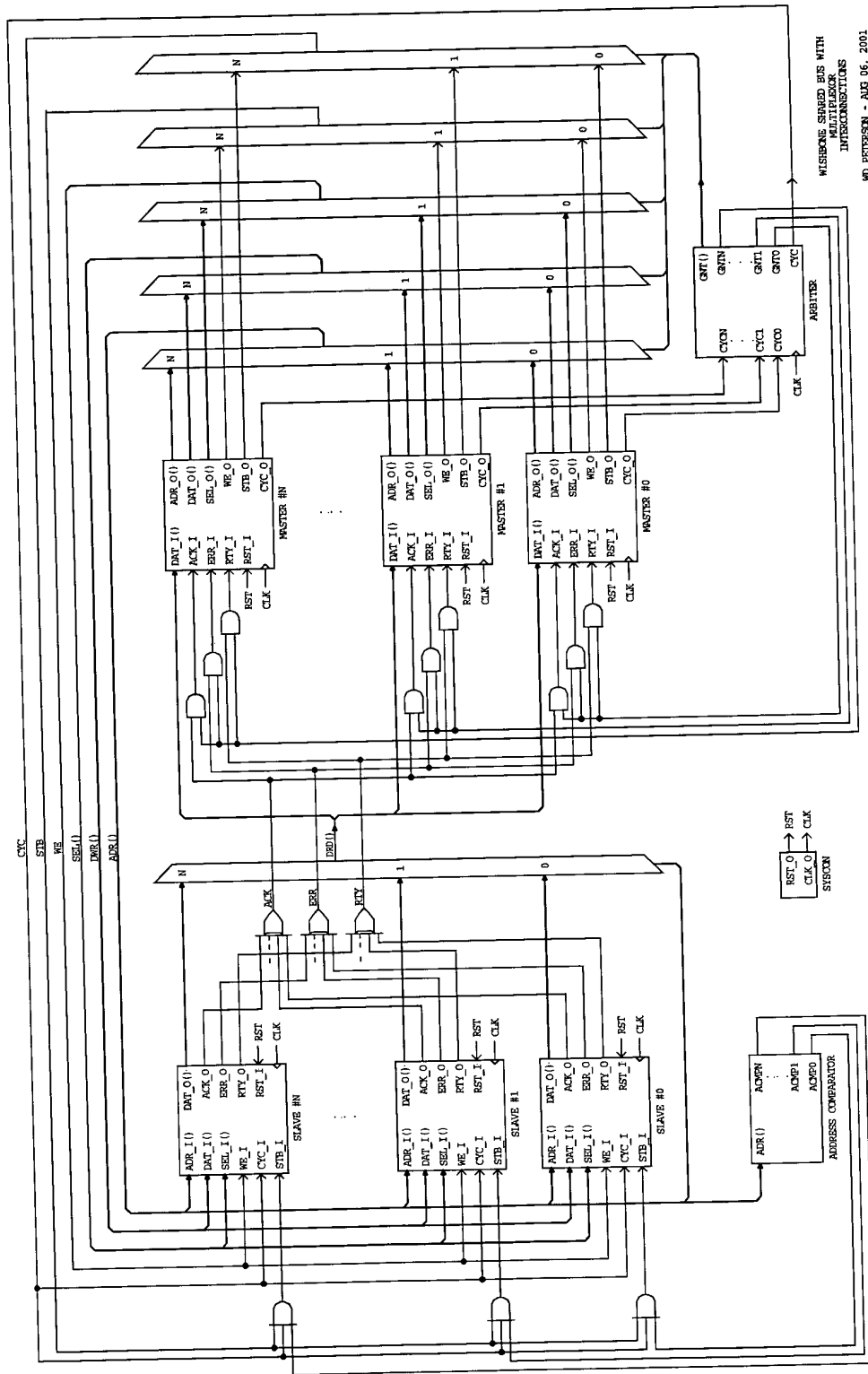
Figure A-22. WISHBONE shared bus with multiplexor interconnections.

Figure A-23. Address map example.

A.9.4 Full vs. Partial Address Decoding

The address comparitor in our INTERCON example is a good way to explain the concept of WISHBONE partial address decoding.

Many systems, including standard microcomputer buses like PCI and VMEbus, use *full address decoding*. Under that method, each slave module decodes the full address bus. For example, if a 32-bit address bus is used, then each slave decodes all thirty-two address bits (or at least a large portion of them).

SoC buses like WISHBONE use *partial address decoding* on slave modules. Under this method, each slave decodes only the range of addresses that it uses. For example, if the slave has only four registers, then the WISHBONE interface uses only two address bits. This technique has the following advantages:

o   It facilitates high speed address decoders.
o   It uses less redundant address decoding logic (i.e. fewer gates).
o   It supports variable address sizing (between zero and 64-bits).
o   It supports the variable interconnection scheme.
o   It gives the system integrator a lot of flexibility in defining the address map.

For example, consider the serial I/O port (IP core) with the internal register set shown in Figure A-24(a). If *full address decoding* is used, then the IP core must include an address decoder to select the module. In this case, the decoder requires: 32 bits – 2 bits = 30 bits. In addition, the IP core would also contain logic to decode the lower two bits which are used to determine which I/O registers are selected.

If *partial address decoding* is used, then the IP core need only decode the two lower address bits ($2^2 = 4$). The upper thirty bits are decoded by logic outside of the IP core. In this case the decoder logic is shown in Figure A-24(b).

Standard microcomputer buses always use the full address decoding technique. That's because the interconnection method does not allow the creation of any new signals on the interface. However, in WISHBONE this limitation does not exist. WISHBONE allows the system integrator to modify the interconnection logic and signal paths.

One advantage of the partial address decoding technique is that the size of the address decoder (on the IP core) is minimized. This speeds up the interface, as decoder logic can be relatively slow. For example, a 30-bit full address decoder often requires at least 30 XOR gates, and a 30-input AND gate.

Another advantage of the partial address decoding technique is that less decoder logic is required. In many cases, only one 'coarse' address decoder is required. If full address decoding is used, then each IP core must include a redundant set of address decoders.

Another advantage of the partial address decoding technique is that it supports variable address sizing. For example, on WISHBONE the address path can be any size between zero and 64-bits. Slave modules are designed to utilize only the block of addresses that are required. In this case, the full address decoding technique cannot be used because the IP core designer is unaware of the size of the system address path.

Another advantage of the partial address decoding technique is that it supports the variable interconnection scheme. There, the type of interconnection logic is unknown to the IP core designer. The interconnection scheme must adapt to the types of slave IP cores that are used.

The major disadvantage of the partial address decoding technique is that the SoC integrator must define part of the address decoder logic for each IP core. This increases the effort to integrate the IP cores into the final SoC.
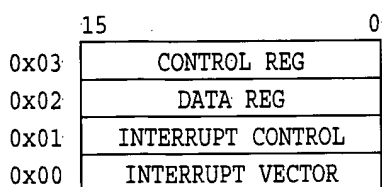
```
            15                      0
         ┌─────────────────────────┐
0x03     │       CONTROL REG       │
         ├─────────────────────────┤
0x02     │        DATA REG         │
         ├─────────────────────────┤
0x01     │     INTERRUPT CONTROL   │
         ├─────────────────────────┤
0x00     │     INTERRUPT VECTOR    │
         └─────────────────────────┘

        (A) SAMPLE IP CORE REGISTER SET
```
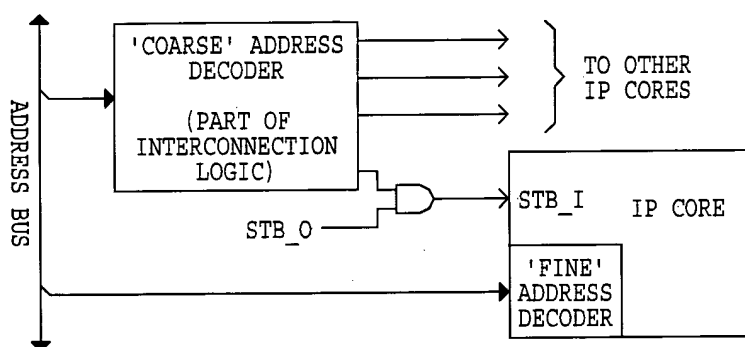

(B) IP CORE ADDRESS DECODING

Figure A-24. WISHBONE partial address decoding technique.

A.9.5 The System Arbiter

The system arbiter determines which MASTER can use the shared bus. The WISHBONE specification allows a variety of arbiters to be used. However, in this example a four level round-robin arbiter is used.

Round-robin arbiters give equal priority to all of the MASTERs. These arbiters grant the bus on a rotating basis much like the four position rotary switch shown in Figure A-25. When a MASTER relinquishes the bus (by negating its [CYC_O] signal), the switch is turned to the next position, and the bus is granted to the MASTER on that level. If a request is not pending on a certain level, the arbiter skips over that level and continues onto the next one. In this way all of the MASTERs are granted the bus on an equal basis.

```
                    MASTER #0

                       ⚲
                       ↑
                       |   ↘
                       |    ↘
    MASTER #3  ○       ○      ○  MASTER #1


                       ○
                   MASTER #2
```
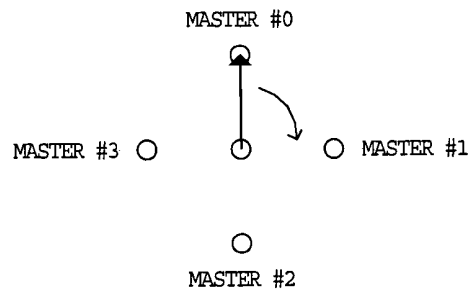
Figure A-25. Round-robin arbiters grant the bus on a rotating
basis much like a rotary switch.

Round-robin arbiters are popular in data acquisition systems where data is collected and placed into shared memory. Often these peripherals must off-load data to memory on an equal basis. Priority arbiters (where each MASTER is assigned a higher or lower level of priority) do not work well in these applications because some peripherals would receive more bus bandwidth than others, thereby causing data 'gridlock'.

The arbiter used in this example can be found in the WISHBONE Public Domain Library for VHDL. ARB0001a is used for the example.

A.9.6 Creating and Benchmarking the System

The final task in our shared bus system example is to create and benchmark the entire system. The INTERCON in our example system is based on the generic shared bus topology that was described above. However, that system is fine tuned to give the exact features that we will need.

The final system supports four DMA0001a MASTERs, four MEM0002a memories (SLAVEs), a 32-bit data bus, a five bit address bus, a single SYC0001a system controller and a ARB0001a

four level round-robin arbiter. The resulting VHDL file can be found under ICN0002a in the WISHBONE Public Domain Library for VHDL.

In this application, the round-arbiter was chosen because all of the MASTERs are DMA controllers. That means that all four MASTERs continuously vie for the bus. If a priority arbiter were used, then only the one or two highest priority MASTERs would ever get the bus.

As we'll see shortly, the error and retry signals [ERR_I] and [RTY_I] are not supported on the MASTER and SLAVE interfaces on our example system. That's perfectly okay because these signals are optional in the WISHBONE specification. We could have added these signals in there, but they would have been removed by synthesis and router logic minimization tools.

Since all of the MASTERs and SLAVEs on this system have identical port sizes and granularities, the select [SEL] interconnection has been omitted. This could have been added, but it wasn't needed.

The INTERCON system includes a partial address decoder for the SLAVEs. This decoder creates the system address space shown in Figure A-26. The final address map is shown in Table A-6.

```
0x1F ┌──────────┐
     │ SLAVE  3 │
0x18 ├──────────┤
0x17 │          │
     │ SLAVE  2 │
0x10 ├──────────┤
0x0F │          │
     │ SLAVE  1 │
0x08 ├──────────┤
0x07 │          │
     │ SLAVE  0 │
0x00 └──────────┘
```

Figure A-26. Address map used by the INTERCON example.

Table A-6. Address spaces used by INTERCON.			
DMA Master:	DMA's To:	At Addresses	Using Cycles
MASTER #0	SLAVE #0	0x00 – 0x07	BLOCK READ/WRITE
MASTER #1	SLAVE #1	0x08 – 0x0F	BLOCK READ/WRITE
MASTER #2	SLAVE #2	0x10 – 0x17	BLOCK READ/WRITE
MASTER #3	SLAVE #3	0x18 – 0x1F	SINGLE READ/WRITE

Source code for this example can be found in the WISHBONE Public Domain Library for VHDL (in the EXAMPLES folder). The library also has detailed documentation for the library

modules, including detailed circuit descriptions and timing diagrams. The reader is encouraged to review and experiment with all of these files.

This system was synthesized and routed on two styles of Xilinx[15] FPGA: the Spartan 2 and the Virtex 2. For benchmarking purposes the memories were altered so that they used Xilinx distributed RAMs instead of the register RAMs in MEM0002a. A memory interface for the Xilinx RAMs can be found in MEM0001a, which is substituted for MEM0002a.

It should be noted that the Xilinx distributed RAMs are quite efficient on the WISHBONE interface. As can be seen in the source code, only a single 'AND' gate was needed to interface the RAM to WISHBONE.

The system for the Xilinx Spartan 2 was synthesized and operated on a Silicore evaluation board. In order to verify that the system actually does run correctly, an HP54620a logic analyzer was connected to test pins on the board, and some of the signals were viewed. Figure A-27 shows the trace. Address lines, data write lines and several control signals are shown.
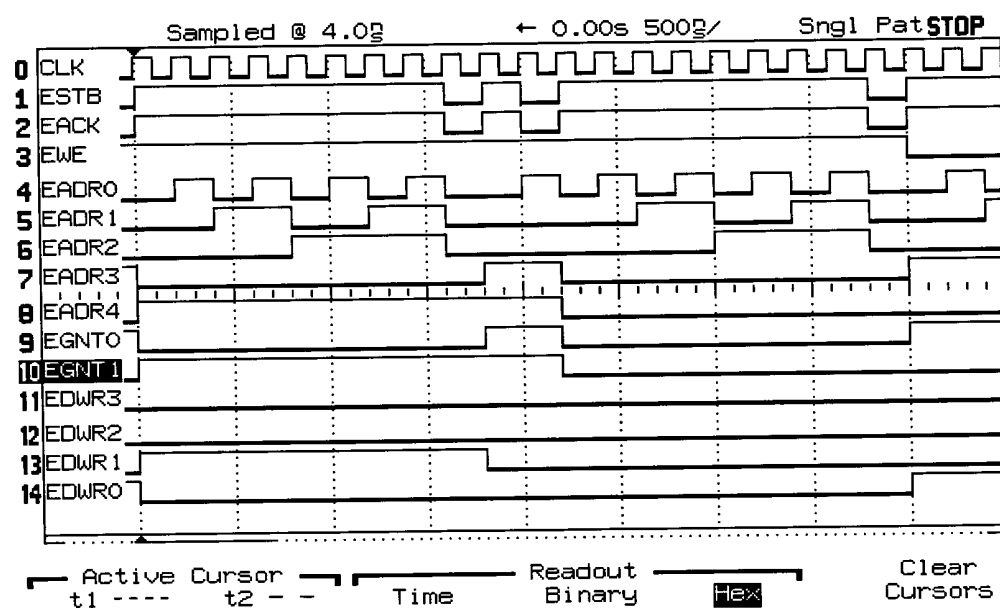
Figure A-27. Logic analyzer trace on the Spartan 2 evaluation board[16].

Table A-7 shows the speed of the system after synthesis and routing. The Spartan 2 benchmarked at about 220 Mbyte/sec, and was tested in hardware (HW TEST). The Virtex 2 part was only synthesized and routed, and showed a maximum speed of about 404 Mbyte/sec (SW TEST).

---

[15] Xilinx is a registered trademark of Xilinx, Inc.
[16] The logic analyzer samples at 500 Mhz, so the SoC was slowed down to make the traces look better. This trace was taken with a SoC clock speed of 5 MHz.

Table A-7. 32-bit Shared Bus Interconnection Benchmark Results							
MFG & Type	Part Number	HW TEST	SW TEST	Size	Timing Constraint (MIN)	Actual Speed (MAX)	Data Transfer Rate (MAX)
Xilinx Spartan 2 (FPGA)	XC2S50-5-PQ208C	√		356 SLICE	55 MHz	55MHz	220 Mbyte/sec
Xilinx Virtex 2 (FPGA)	XC2V250-5-FG256C		√	355 SLICE	99 MHz	101 MHz	404 Mbyte/sec

Notes:
VHDL synthesis tool: Altium Accolade PeakFPGA 5.30a
Router: Xilinx Alliance 3.3.06I_V2_SE2
Hardware evaluation board: Silicore 170101-00 Rev 1.0
Listed size was reported by the router.
Spartan 2 test used '-5' speed grade part (slower than the faster '-6' part).

A.10 References

Di Giacomo, Joseph. <u>Digital Bus Handbook</u>. McGraw-Hill 1990. ISBN 0-07-016923-3.

INDEX

Ω

Media Release

For immediate release

Altium unveils new "Board-on-Chip" technology

Altium to preview industry-first technology at Programmable World that allows engineers to use board-level methodologies to design *and* implement embedded systems on FPGAs

SYDNEY, Australia – April 28, 2003 – Altium Limited (ASX: ALU), a leading developer of Windows-based electronic design and development software, announced today that it will preview a new design technology that brings together the worlds of software and hardware design in a single, integrated design environment. This new technology enables engineers to use board-level design methodologies to both design *and* implement an entire digital system – including embedded microprocessor cores and the software that runs on them – onto an FPGA. Altium claims that this new technology gives engineers the power to literally start putting their board designs straight into FPGAs, a process that Altium calls "Board-on-Chip" (BoC) design.

Despite the emergence of high capacity, low cost FPGAs that offer the potential to be used as a platform for digital system design, there are currently significant cost and technology barriers facing engineers who wish to exploit this potential. Current tools are primarily HDL based and not well integrated with embedded software tools. Also, HDL-based IP cores are expensive and have complex licensing schemes. To date, these barriers have hindered the penetration of FPGAs as a platform solution for mainstream engineers.

Altium says that its new BoC technology will enable engineers to overcome these barriers and use familiar design methodologies to harness the power of FPGAs as a design platform. "*We recognize the potential of FPGAs as a design platform but currently tools to allow engineers to fully exploit this potential are just not there,*" says Nick Martin, Joint CEO and founder, Altium. "*However, the technology that we are developing at Altium will allow us to deliver a system that enables rapid implementation and testing of complete digital systems on FPGAs – without the need for HDL source files or extensive synthesis and verification.*"

Board-on-Chip technology

In order to facilitate BoC design, Altium has brought together technology from the full range of its EDA and embedded products. Altium's nVisage multi-dimensional design capture and TASKING embedded software technologies (including Viper multi-core compiler and debugger technologies) have been combined on the Design Explorer (DXP) platform to provide a single, integrated hardware design and software development environment.

The BoC system that Altium will preview at Programmable World will include:

- **Mixed schematic/VHDL** design capture
- Integrated **software development**
- **Processor core packs** that combine pre-synthesised processor cores with matching compiler, simulator and debugger
- **Schematic component libraries** containing a range of pre-synthesised components including common peripheral devices, 74xxx series logic and a range of communication and interface components
- **Primitives and macro libraries** for all Xilinx and Altera device families
- **'Virtual' instruments** such as logic analyzers and frequency counters that can be built into the design for test purposes
- A **BoC development board** loaded with an FPGA that functions as a breadboard and allows implementation of the design directly from the engineer's PC onto the FPGA.

"What makes this technology a real break-through is that it will provide a fully-integrated and self-contained system for FPGA-based design that will be accessible to the vast majority of engineers," says Martin. *"This BoC technology represents a new EDA paradigm and has the potential to significantly lower development costs and provide huge benefits to engineers in terms of time to market and design flexibility."*

True parallel development of hardware and software

One of the most significant benefits of Altium's BoC technology is that it allows a more flexible approach to partitioning the design between hardware and software – engineers retain the ability to choose a hardware or software solution to any particular problem throughout the design process. Also, the inclusion of a targeted development board as part of the BoC system allows hardware dependencies in the software to be addressed before the target hardware is finalized.

In general software tends to remain fluid throughout the development process because it is easy to update and change. Hardware, on the other hand, tends to be locked down early in the process because of time and cost involved in each hardware iteration and the fact that the board needs to be available in

order to finalize software development. However, using the BoC system with its reconfigurable FPGA-based hardware platform, hardware updates can be made as easily as software updates. This makes it possible to continue to explore different hardware options throughout the design process and allows final hardware/software partitioning decisions to be delayed until late in the development process.

Reducing development costs and time to market

The more flexible and parallel design approach enabled by the BoC technology will dramatically reduce time to market compared to traditional linear process flows. Within the BoC system, a new prototype is effectively generated each time the design is downloaded to the development board. This means that systems can be more fully developed and tested before fabricating the first PCB prototype. Also, because more of the design is implemented inside the FPGA, including the processor, the final PCB design and fabrication will be simpler, leading to further cost and time savings.

Further information

Altium will be demonstrating its BoC system at the Programmable World forum which begins on May 6. For details go to www.xilinx.com/events/pw2003/index.htm.

About Altium Limited

Altium Limited (ASX: ALU), trading as Protel International Limited (ASX: PRI) prior to August 6, 2001, is a leading global developer and supplier of desktop Electronic Design Automation (EDA) and embedded software design tools for the Microsoft Windows environment.

Since the company's foundation in 1985 and its release of the world's first Microsoft Windows–based EDA tool in 1991, Altium has continued to apply the most advanced software design methods to provide powerful, easy-to-use and affordable design software to engineers and electronics designers worldwide.

Altium's current product brands include Protel, nVisage, P-CAD, TASKING, Accolade, CircuitMaker and CAMtastic. These products offer tailored solutions covering a range of hardware and software design processes.

Altium is headquartered in Sydney, Australia, and operates a number of sales and support offices in Australia, the United States, Japan and Europe, as well as maintaining a large reseller network in all other major markets. More information about the company and its products and services may be obtained from our website at www.altium.com.

THIS PAGE BLANK (USPTO)

Media Release

Altium
Making Electronics Design Easier™

Altium introduces systems focus to FPGA design with Nexar

Altium releases industry's first out-of-the-box design environment for putting entire embedded systems on FPGAs

SYDNEY, Australia – November 17, 2003 – Altium Limited (ASX: ALU), a leading developer of Windows-based electronics design software, today announced Nexar, the industry's first product to provide a comprehensive, vendor-independent solution for system-level design on an FPGA platform. Derived from Altium's demonstrated Board-on-Chip technology, Nexar integrates hardware design tools, embedded software development tools, IP-based components, virtual instrumentation and a reconfigurable development board to allow mainstream engineers, even those without HDL experience, to interactively design and implement a complete embedded system inside an FPGA.

The benefits Nexar brings to engineers include parallel design of hardware and software, greater flexibility in hardware/software design partitioning, an integrated, FPGA vendor-independent solution for putting entire embedded systems into FPGAs, and a "live", interactive design environment for system-on-FPGA development and debug.

"Nexar is the first complete system-on-FPGA design environment built upon 'live' real-time, hands-on engineering that happens inside the physical hardware design space using an approach that maps directly to an engineer's existing knowledge of system-level design," says Nick Martin, Altium's founder and Joint CEO. *"The Nexar environment will be delivered complete and ready to use, with design hardware, design software and IP all in one box. Nexar will unlock the potential of current and next-generation high-capacity, low-cost FPGAs for every engineer."*

While many engineers look to FPGA technology to provide higher levels of on-chip integration and a lower risk alternative to the cost and lead time of conventional ASICs, system-level design on an FPGA platform is a difficult exercise, particularly when it comes to bringing the processor into the FPGA. Nexar changes this by taking proven board-level system design methodologies and retargeting them for FPGA

architectures. Nexar also integrates hardware design and software development within a single environment to provide a total solution to systems design.

The result is a revolutionary system-on-FPGA product that will allow risk-free chip-level systems integration, practical hardware/software co-design, a complete systems-level development environment for FPGA-based embedded design and introduces an interactive design methodology called LiveDesign that is accessible to mainstream engineers.

Introducing a new IP delivery model

At the system level, Nexar provides a schematic-based design methodology to define system connectivity. The reason being that graphical schematic-based capture is more efficient for connecting functional blocks than HDLs and allows complex systems to be created quickly at the component level.

Schematic design is facilitated in Nexar by the inclusion of extensive libraries of royalty-free, pre-synthesized, pre-verified IP components, including a range of processor cores, that can be simply dropped onto the schematic and connected together to form the system hardware. This is analogous to the way designers currently work at the board-level with physical, 'off-the-shelf' components.

Nexar components are processed for a variety of target FPGA architectures from multiple vendors. This allows design portability between FPGA device families and ensures a flexible vendor-independent approach to FPGA design. Nexar automatically and transparently selects the correct component model for the target architecture during system synthesis. As a result, designs can be synthesised very quickly because the component IP does not need to be reprocessed during synthesis.

Nexar's component system provides a novel and secure framework for FPGA IP delivery that avoids the security problems associated with supplying IP as HDL source code.

Seeing inside the FPGA with virtual instrumentation

Along with IP components, Nexar includes a library of IP-based virtual instruments such as logic analysers, frequency counters/generators and IO monitors that can be incorporated into the design at the schematic level to facilitate system debugging.

Like the IP components, the virtual instruments are supplied as pre-synthesized models that allow them to be used across FPGA target architectures. These instruments have on-screen front panels analogous to their physical counterparts to provide an intuitive way for engineers to examine the working of their circuit, and to 'see' inside the FPGA during the design process.

A reconfigurable hardware platform

Integral to Nexar is a versatile FPGA-based development board called a NanoBoard that provides a reconfigurable platform for implementing and debugging the design. The NanoBoard is connected to the engineer's PC and uses JTAG-based communications to both download the design to the on-board FPGA, and to interact with processor cores and instruments in the design once it has been downloaded.

Target FPGAs are housed on plug-in daughter boards to allow easy retargeting of designs. Multiple NanoBoards can be chained together to facilitate the design of complex multi-FPGA systems, and can accommodate the inclusion of end-user boards into the system for final production PCB testing and debugging.

Integrated software development

To make it easy to develop system software, Nexar includes a complete set of software development tools for all supplied processor cores. Using Altium's Viper reconfigurable compiler technology, Nexar provides high-quality code development and debugging that is fully integrated with the hardware development environment.

Once the target design has been downloaded to the NanoBoard, all processors in the design can be controlled and debugged from within the Nexar system. This enables software development to take place directly on the target hardware from early in the design cycle, supporting parallel development of hardware and software. Hardware designers can download their designs to the NanoBoard for interactive debugging during development, and software designers can develop their program code directly on the 'real' hardware from early in the design cycle.

Because hardware can be updated with the same ease as the software, Nexar allows more flexible design partitioning. Implementing the design on the NanoBoard means a physical prototype isn't required to support completion of the debug process, delaying the need to finalize hardware until later in the development cycle.

LiveDesign environment

Nexar's interactive system design environment and ability to directly connect designers and developers with their designs allows engineers to adopt a very 'hands on' approach to the development process. Altium calls this design methodology LiveDesign.

Unlike conventional electronics design flows, LiveDesign-enabled tools represent a new methodology for hardware/software system design flows. The LiveDesign electronics development platform supports real-time on-the-fly design and debug of a physical circuit and has the potential to significantly effect the way electronic systems and products are designed with benefits for design speed, flow, quality and cost.

LiveDesign also allows system implementation for debug purposes, minimizing the reliance on time-consuming system-level simulation required by other design flows. There is little time or cost penalty involved in multiple design iterations, leaving engineers free to try different design solutions without the need to physically manufacture prototype boards.

Nexar benefits

Nexar has wide applicability across the electronics industry, and will be particularly beneficial in industries such as automotive, industrial control, telecoms and datacoms infrastructure, and non-electronics consumer products such as whitegoods, where product value is relatively high, but market size doesn't allow conventional ASIC development.

Nexar will provide immediate benefits to:

- Engineers who routinely develop digital systems using off-the-shelf silicon devices – Nexar provides an intuitive method for migrating board-level systems into device-level designs;
- FPGA designers looking for an easier way to embed soft processor cores into their designs – Nexar provides a reconfigurable development environment, the tools and IP to support complete systems implementation;
- Hardware & software engineers designing low- to medium-complexity embedded microcontroller applications – Nexar integrates hardware and software development flows from the first line of code through board-level implementation.

Nexar represents a new way of approaching digital design. In particular Nexar will provide:

- Shorter embedded systems development cycles by enabling parallel design of hardware and software;
- Greater flexibility in hardware software design partitioning;
- An integrated solution for putting entire embedded systems into FPGAs;
- The benefits of chip-level integration to all designers, regardless of their HDL knowledge and expertise;
- A personal ASIC alternative to high-cost factory ASIC implementation;
- An FPGA vendor-independent solution to systems design;
- A "live", interactive environment for system-on-FPGA development and debug.

Pricing and availability

Nexar 2004 will be available for shipment during the first quarter of the 2004 calendar year at an anticipated list price of US$7,995. For more product information, visit www.altium.com/nexar/ or contact your local Altium sales and support center.

THIS PAGE BLANK (USPTO)

About Altium Limited

Altium Limited (ASX: ALU) is a global developer and supplier of electronics design software for the Microsoft Windows environment. Founded in 1985, Altium released the world's first Microsoft Windows-based printed circuit board design tool in 1991 and continues to provide advanced, easy-to-use and affordable software design tools to electronics engineers, designers, and developers worldwide. Altium's products offer tailored solutions covering a range of hardware and software design processes including the well-known Protel, P-CAD and TASKING brands. Altium is headquartered in Sydney, Australia and has sales and support offices in Australia, the United States, Japan and Europe. More information is available at www.altium.com.

Altium, Nexar, LiveDesign, NanoBoard, Protel, DXP, Design Explorer, nVisage, PCAD, TASKING, Accolade, CircuitMaker, CAMtastic, Situs and Topological Autorouting and their respective logos are trademarks or registered trademarks of Altium Limited or its subsidiaries. All other registered or unregistered trademarks referenced herein are the property of their respective owners, and no trademark rights to the same are claimed.

THIS PAGE BLANK (USPTO)